



# ARM Software Development Toolkit Version 2.11

## User Guide

Document number: ARM DUI 0040C

Issued: May 1997

Copyright Advanced RISC Machines Ltd (ARM) 1996

### ENGLAND

Advanced RISC Machines Limited  
Fulbourn Road  
Cherry Hinton  
Cambridge CB1 4JN  
UK

Telephone: +44 1223 400400  
Facsimile: +44 1223 400410  
Email: info@arm.com

### JAPAN

Advanced RISC Machines K.K.  
KSP West Bldg, 3F 300D, 3-2-1 Sakado  
Takatsu-ku, Kawasaki-shi  
Kanagawa  
213 Japan

Telephone: +81 44 850 1301  
Facsimile: +81 44 850 1308  
Email: info@arm.com

### GERMANY

Advanced RISC Machines Limited  
Otto-Hahn Str. 13b  
85521 Ottobrunn-Riemerling  
Munich  
Germany

Telephone: +49 89 608 75545  
Facsimile: +49 89 608 75599  
Email: info@arm.com

### USA

ARM USA Incorporated  
Suite 5  
985 University Avenue  
Los Gatos  
CA 95030 USA

Telephone: +1 408 399 5199  
Facsimile: +1 408 399 8854  
Email: info@arm.com

World Wide Web address: <http://www.arm.com>

---

## Proprietary Notice

ARM, the ARM Powered logo and EmbeddedICE are trademarks of Advanced RISC Machines Ltd.

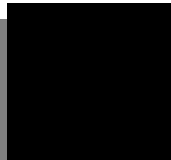
Neither the whole nor any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this manual is subject to continuous developments and improvements. All particulars of the product and its use contained in this manual are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties or merchantability, or fitness for purpose, are excluded.

This manual is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this manual, or any error or omission in such information, or any incorrect use of the product.

## Change Log

Issue	Date	By	Change
A-xx	November 1996	JS	Updated for Beta Test
A	December 1996	Doc Group	Final draft
B	January 1997	JS	Review comments added
C	May 1997	paw	general update



# Preface

About This Manual  
Feedback

Preface-2  
Preface-3



# Preface

---

## About This Manual

### Overview

This manual describes the ARM Software Development Toolkit (SDT), a suite of applications which, together with supporting documentation and examples, enables you to write and debug applications for the ARM family of RISC processors.

### Typographical Conventions

The following typographical conventions are used in this book:

<code>typewriter</code>	denotes text that may be entered at the keyboard: commands, file and program names, and C source
<code>typewriter-italic</code>	shows text that must be submitted with user-supplied information: this is most often used in syntax descriptions
<i>italic</i>	highlights important notes and ARM-specific terminology
<b>bold</b>	signal names and menu selections
<b><i>bold</i></b>	internal cross-references





## Feedback

### Feedback on this manual

If you have any comments or suggestions about this document, please contact your supplier giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- a concise explanation of your comments

### Feedback on the Software Development Toolkit

If you have any problems with the Software Development Toolkit, please contact your supplier including the following information:

- details of the version of tools you are using, and the host system
- a small sample code fragment that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened





# Contents

<b>1</b>	<b>Introduction</b>	<b>1-1</b>
1.1	Introduction to the ARM Software Development Toolkit	1-2
1.2	Components of the SDT	1-3
1.3	What's New	1-5
<b>2</b>	<b>ARM Project Manager</b>	<b>2-1</b>
2.1	Introduction	2-2
2.2	APM Concepts	2-3
2.3	The APM Desktop	2-9
2.4	Getting Started	2-13
2.5	Additional APM Functions	2-20
2.6	Project Manager Configuration	2-26
2.7	Working with Source Files	2-28
2.8	Viewing Object and Executable Files	2-31
2.9	Working with Project Templates	2-33
2.10	Build Step Patterns	2-41
<b>3</b>	<b>ARM Debugger for Windows</b>	<b>3-1</b>
3.1	Introduction	3-2

# Contents

---

3.2	Debugging an ARM Application	3-3
3.3	Debugging Systems	3-3
3.4	ADW Concepts	3-6
3.5	The ADW Desktop	3-11
3.6	Getting Started	3-21
3.7	Debugger Configuration	3-28
3.8	Displaying Image Information	3-34
3.9	Setting and Editing Complex Breakpoints and Watchpoints	3-39
3.10	Other Debugging Functions	3-42
<b>4</b>	<b>Command-Line Development</b>	<b>4-1</b>
4.1	Introduction	4-2
4.2	The Hello World Example	4-2
4.3	armsd	4-7
<b>5</b>	<b>The ARMulator</b>	<b>5-1</b>
5.1	What is the ARMulator?	5-2
5.2	Models	5-2
5.3	Rebuilding the ARMulator	5-4
5.4	ARMulator facilities	5-6
5.5	Supplied Models: Angel	5-9
5.6	Supplied Models: Dummy MMU	5-11
5.7	Supplied Models: Profiler	5-12
5.8	Supplied Models: Tracer	5-13
5.9	Supplied Models: Watchpoints	5-15
5.10	Supplied Models: Windows Hourglass	5-15
5.11	Supplied Models: Page Table Manager	5-15
5.12	An Example Memory Model	5-17
<b>6</b>	<b>Angel</b>	<b>6-1</b>
6.1	Introduction	6-2
6.2	Overview of Developing Applications with Angel	6-4
6.3	Inside Angel	6-9
6.4	Angel System Features	6-12
6.5	Debuggers	6-16
6.6	Minimal Angel	6-17
6.7	Porting Angel to New Hardware	6-18
6.8	Downloading a Debug Agent	6-21
6.9	Adding a Channel	6-22
6.10	Using Angel on the ARM PID7T board	6-23
6.11	Using Angel on the ARM60 PIE Board or ARM7 PIE Board	6-26
6.12	The Debug Comms Channel, Angel, and EmbeddedICE	6-27
6.13	Notes for Demon Users	6-29
6.14	Example of an Application Device using Angel (UNIX only)	6-31

<b>7</b>	<b>EmbeddedICE</b>	<b>7-1</b>
7.1	What is EmbeddedICE?	7-2
7.2	The Effect of EmbeddedICE on the Debuggee	7-5
7.3	Connecting and Powering Up	7-6
7.4	Configuring EmbeddedICE to Match your Target Core	7-7
7.5	Configuring the Debugger	7-9
7.6	Watchpoints and Breakpoints	7-13
7.7	Vector Breakpoints and Exceptions	7-15
7.8	Semihosting	7-17
7.9	Reset and JTAG Signal Connection	7-20
7.10	Debugging Applications in ROM	7-24
7.11	Accessing the EmbeddedICE Macrocell Directly	7-28
7.12	EmbeddedICE and Target Board Memory Layout	7-30
7.13	Timer Accuracy	7-32
7.14	Floating-Point and Other Coprocessors	7-39
<b>8</b>	<b>Benchmarking, Performance Analysis and Profiling</b>	<b>8-1</b>
8.1	Introduction	8-2
8.2	Measuring Code and Data Size	8-3
8.3	Performance Benchmarking	8-6
8.4	Improving Performance and Code Size	8-14
8.5	Profiling	8-19
<b>9</b>	<b>Basic Assembly Language Programming</b>	<b>9-1</b>
9.1	Introduction	9-2
9.2	Structure of an ARM Assembler Module	9-5
9.3	Assembler Subroutines	9-7
9.4	Structure of a Thumb Assembler Module	9-8
9.5	Loading Constants into Registers	9-10
9.6	Conditional Execution	9-14
9.7	Loading Addresses into Registers	9-19
9.8	Calling Assembler from C	9-25
9.9	Load and Store Multiple Registers Instructions	9-27
<b>10</b>	<b>Using the Procedure Call Standards</b>	<b>10-1</b>
10.1	Introduction	10-2
10.2	Using APCS	10-3
10.3	Using the Thumb Procedure Call Standard	10-9
10.4	Passing and Returning Structures	10-11
<b>11</b>	<b>Exception Handling</b>	<b>11-1</b>
11.1	Overview	11-2
11.2	Entering and Leaving an Exception	11-5
11.3	The Return Address and Return Instruction	11-6

# Contents

---

	11.4	Installing an Exception Handler	11-8
	11.5	SWI Handlers	11-12
	11.6	Interrupt Handlers	11-19
	11.7	Reset Handlers	11-25
	11.8	Undefined Instruction Handlers	11-26
	11.9	Prefetch Abort Handler	11-27
	11.10	Data Abort Handler	11-28
	11.11	Chaining Exception Handlers	11-29
	11.12	Additional Considerations on Thumb-Aware Processors	11-31
	11.13	System Mode	11-35
<b>12</b>		<b>Interworking ARM and Thumb</b>	<b>12-1</b>
	12.1	Introduction	12-2
	12.2	Basic Assembler Interworking	12-4
	12.3	C Interworking and Veneers	12-10
	12.4	Assembler Interworking Using Veneers	12-18
	12.5	Interworking and the ARM Project Manager	12-21
	12.6	Using the Thumb-ARM Interworking Image Project	12-22
	12.7	Modifying Project to Support Interworking	12-24
	12.8	Library usage and the ARM Project Manager	12-25
<b>13</b>		<b>Writing code for ROM</b>	<b>13-1</b>
	13.1	Introduction	13-2
	13.2	Application Startup	13-3
	13.3	The Embedded C Library	13-16
	13.4	Troubleshooting Hints and Tips	13-24
<b>14</b>		<b>Placing Code and Data in Memory</b>	<b>14-1</b>
	14.1	Introduction	14-2
	14.2	Scatter Loading	14-3
	14.3	Overlays	14-15
	14.4	Overlays using Scatter Loading	14-28
<b>15</b>		<b>Floating-Point Support</b>	<b>15-1</b>
	15.1	Introduction	15-2
	15.2	The ARM Floating-point Library	15-3
	15.3	Floating-Point Instructions	15-8
	15.4	Linking the FPE into an Application	15-13
	15.5	Configuring the FPA Support Code/FPE for a New Environment	15-14
	15.6	Controlling Floating-Point Exceptions	15-15

# 1

## Introduction

1.1	Introduction to the ARM Software Development Toolkit	1-2
1.2	Components of the SDT	1-3
1.3.2	What was new in SDT 2.10	1-5

# Introduction

---

## 1.1 Introduction to the ARM Software Development Toolkit

The *ARM Software Development Toolkit (SDT)* consists of a suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for ‘the’ ARM family of RISC cores.

The SDT can be used for two main purposes:

- Software development This involves building either C or ARM assembler source code into ARM object code, which can then be debugged using an ARM symbolic debugger. The debugger has facilities that include single stepping, setting breakpoints and watchpoints, and viewing registers. Testing and debugging can be carried out on code running under software emulation in the host system, or on an ARM core attached to it. The toolkit includes the emulators required to run your applications even when you do not have access to real ARM hardware.
- Benchmarking Once an application has been built, it can be benchmarked on an ARM core, or on a software emulator.





## 1.2 Components of the SDT

The ARM SDT consists of the following command line tools:

<code>armcc</code>	The ARM C cross compiler. This is a mature, industrial-strength compiler, tested against the Plum Hall C Validation Suite for ANSI conformance. It supports both Unix- and PCC-compatible modes, and is able to optimize for code size or execution speed. The compiler is very fast, compiling 500 lines per second on a SPARC 10/41, and can output object code or assembler source.
<code>tcc</code>	The Thumb C cross compiler. This is based on the ARM C compiler but produces 16-bit Thumb instructions instead of 32-bit ARM instructions.
<code>armasm</code>	The ARM cross assembler. This compiles ARM assembly language source into ARM object format object code.
<code>tasm</code>	The Thumb and ARM cross assembler. This compiles both ARM assembly and Thumb assembly language into the equivalent object code (32-bit instructions for ARM, 16-bit instructions for Thumb).
<code>armlink</code>	The Thumb and ARM linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program. armlink creates ARM image format or ELF executable images.
<code>decaof</code>	The Thumb and ARM object file decoder/disassembler. This decodes object files and enables you to extract information, such as code size, from object files.
<code>decaxf</code>	The ARM executable file decoder/disassembler. This is used to extract information from AIF or ELF executable images.
<code>armulator</code>	The ARM core emulator. This emulates ARM cores at the instruction set level, allowing ARM and Thumb executable programs to be run on non-native hardware. The armulator is not provided as a stand-alone program, but is integrated into ARM debuggers.
<code>armsd</code>	The command-line Thumb and ARM symbolic debugger. This enables source level debugging of programs, either running on the integrated armulator, or on an ARM core connected to the host system. Facilities include single stepping through C or assembler source, setting breakpoints and watchpoints, and examining program variables or memory.

These tools are fully documented in the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

# Introduction

---

The ARM Windows development tools additionally include:

- APM**      The ARM Project Manager (APM). This enables you to construct the environment and the procedures necessary to automate the process of building your system. APM schedules the calling of development tools such as compilers, linkers and your own custom tools so that you can bring a whole system up to date with the minimum of effort. See ***1.3.2 What was new in SDT 2.10*** for further details.
- ADW**      The ARM Debugger for Windows. This provides a fully windowed environment for debugging programs. As with armsd, programs can be run on the integrated ARMulator, or on an ARM core attached to a host system.

## 1.3 What's New

### 1.3.1 What's new in SDT 2.11

- ARM and Thumb in line assembler in the C compilers.
- Improved scatter loading in the ARM linker.
- Floating-point support has been improved with the source of the floating-point support code.
- More pre-built variants.
- The documentation set has been improved with the addition of User Guide chapters covering the *ARM Project Manager (APM)* and the *ARM Debugger for Windows (ADW)*.
- APM has been improved with reduced build times.

### 1.3.2 What was new in SDT 2.10

### 1.3.3 Core tools

- ARM8, StrongARM and ARM7500FE cores are now supported by the ARM C compiler and assembler.
- ELF and DWARF file formats are supported.
- Compilation speed has been improved.
- The file size overhead created by compiling with debug has been reduced.
- ARMulator now supports cached cores, including ARM610/710/810 and StrongARM 110.

### 1.3.4 Windows tools

New improved release of APM including:

- Projects are defined by picking a matching template.
- Project files are automatically organized by user-defined variant and partition, making it easy to organize large numbers of source file.
- User-extendable to include custom tools and 'build steps'. Non-ARM utilities and tools can be called to build the project.
- Build projects which include sub-projects—multi-level builds for complex applications.
- 'Tree View' of project structure.
- Integration with 3rd party tools.
- GUI Configuration of ARM core tools.
- Faster build times through improved 'dependency' checking.
- Ability to launch an 'alien' debugger directly from the Project Manager.

# Introduction

---

New improved release of ADW includes:

- Support for the ARM7TDMI communications channel and the PID7T development board
- Ability to program PID7T flash memory directly from the debugger
- Support for ARMulator, EmbeddedICE, and the Angel and (for compatibility) Demon debug monitors

## 1.3.5 Debug monitor

Tools 2.10 introduces the new Angel debug monitor which is a complete replacement for the Demon debug monitor. Angel includes:

- Support for PID7T and EBSA110 development boards.
- Host connection via Serial, Parallel, 7TDMI Comms Channels and Ethernet (option with PID7T).
- ARM- and Thumb-state debugging.
- Written in C for portability.
- Provided as a library for linking into users application.
- Modular, enabling the user to choose minimum level of support to match requirements.

## 1.3.6 Platforms supported

This release of the ARM SDT is supported on a variety of platforms. The command-line development tools are supported on:

- Sun Workstations running SunOS 4.1.3, Solaris 2.4 or Solaris 2.5.
- Hewlett Packard Workstations running HP/UX 9.0.
- IBM Compatible PCs running Windows 95 or Windows NT 3.51 or 4.0.
- Digital Alpha PCs running Windows NT 3.51 or 4.0.

The Windows development tools are supported on:

- IBM Compatible PCs running Windows 95 or Windows NT 3.51 or 4.0.
- Digital Alpha PCs running Windows NT 3.51 or 4.0.

With this release, support for Windows 3.x has been withdrawn.

# 2

## ARM Project Manager

2.1	Introduction	2-2
2.2	APM Concepts	2-3
2.3	The APM Desktop	2-9
2.4	Getting Started	2-13
2.5	Additional APM Functions	2-20
2.6	Project Manager Configuration	2-26
2.7	Working with Source Files	2-28
2.8	Viewing Object and Executable Files	2-31
2.9	Working with Project Templates	2-33
2.10	Build Step Patterns	2-41

# ARM Project Manager

---

## 2.1 Introduction

The *ARM Project Manager (APM)* enables you to construct the environment and the procedures necessary to automate the process of building your software. APM schedules the calling of development tools such as compilers, linkers and your own custom tools so that you can bring a whole project up to date with a minimum of effort.

When you create a project with APM, everything you need to work with is accessible from the APM desktop.

APM uses the concept of a *project* to maintain information about the system you are building. A project describes what to build and how to build it. Once you have described your system as a project, you can build all of it or just a portion of it. If the project output is an image, you can execute it or debug it without leaving the Project Manager.

APM doesn't control how a file is built. That information is supplied by the *project template*, which means you have control over the build tools (compilers, assemblers linkers etc.) that are used and you have control over the command line options that are used by the build tools.

This chapter provides:

- an introduction to APM
- a definition of the terms associated with APM
- an overview of the APM desktop and windows
- a step by step guide to creating, building and running a simple project
- an overview of APM configuration
- an overview of working with source files
- information on displaying binary files
- a step-by-step guide to editing and creating templates and template elements

For detailed instructions on how to use the APM, refer to the comprehensive on-line help.

### 2.1.1 On-line help

When you have started the APM, you can use on-line help to quickly find information on the tasks you are performing. You have several options for accessing the Help system:

- |                 |   |
|-----------------|---|
| <b>Contents</b> | Select <b>Contents</b> from the <b>Help</b> menu to display a Table of Contents.  |
| <b>Search</b>   | Select <b>Index</b> from the <b>Help</b> menu to display an index of all the help topics.                               |
| <b>Help</b>     | Click the <b>Help</b> button to get information on the dialog currently on display.                                     |
| <b>F1</b>       | Press the <b>F1</b> key on your keyboard to get help on the currently active window or the dialog currently on display. |

## 2.2 APM Concepts

### 2.2.1 Projects and sub-projects

A project is a “recipe” that describes how to build something (typically an image or object library) and the list of the constituent files (source files, include files, sub-projects etc.) needed to build the project.

APM defines the instructions and the “something” (the project output) using a project template. A template is made up of build step patterns that define the build steps used when you construct your project’s output.

The source files and files generated by the build steps are organized into a project hierarchy using partitions. APM uses variants to create different versions of your project output(s), such as a debug version and a release version (the defaults).

Project templates give you a great deal of flexibility when you build your project output. You can use a project template provided with APM or you can construct your own templates. You can modify a project template by adding tools and changing how they are executed within the building of the project. You can modify variables at any level in the project hierarchy to change how specific files are handled in the various build steps. You can also create additional variants to meet your project’s requirements.

A sub-project is simply an APM project that has been added to another project. For example if you have a project that builds a library, it could become a sub-project of a project that makes an image using routines from that library. In the basic templates supplied with APM, sub-projects are added to the SubProjects partition (**2.2.8 Partitions** on page 2-6).

### 2.2.2 Project files

Source files are the basic building blocks of a project. The default APM templates recognize the following source file types. The symbols are used to represent them in the Project Window are:



A C source file.



An ARM assembler file.



An include file.



A sub-project. This is also a type of source file, contributing its own project output to the current project.



File type is not known to APM.

When a source file is added to a project, it is not moved from its original location, but rather its location is referenced from the project file. APM tries to refer to files relative to the project directory structure rather than absolutely. You can set the variable `$$DepthOfDotAPJBelowProjectRoot` to increase the scope of the directories that are considered a part of the project.

**Note** *If a project might be moved from its original location, the directory structure containing it must remain consistent, or files required to build the project cannot be found by APM.*

# ARM Project Manager

---

Derived files are created as the result of a build step, such as a compile or a link. The following Symbols are used to represent derived files:



An object file.



A library.



An ARM executable image.

## 2.2.3 Build

A build (or force build) is the processing of the files in a selected variant through a number of build steps. A build or force build executes the build steps required to create the project's output. A build executes a build step only if some input to it is newer than its outputs; a force build executes all build steps.

The actions performed in the various build steps and the nature of the project output(s) are determined by the project's template. The project template also partially determines the build order (eg. Compile steps must take place before Link steps).

Once you have added the necessary files to the project, you can build the project. As your project is being built, the progress indicator at the bottom of the Project Window is updated and any diagnostic information generated is displayed in the Build Log.

The simplest way to turn your source files into your project output is to build the entire project. You can also build a single source file, force the build of the entire project, or select multiple variants to be built.

### Building a project

When you build a project, only the files that are older than the files they depend on are rebuilt and only one variant is built; by default this is the Debug Variant. To build the entire project, select **Build *project-name*** from the Project Menu or click the **Build** button. When you build a project only the active variant is built. If the appropriate setting is selected, **Build** also builds any sub-projects associated with the projects (see **2.6.1 APM preferences** on page 2-26).

### Building a single source

You can build a single file if it is associated with a project and has been opened as a part of a project. If the file can be built (eg. compiled or assembled) the appropriate menu item in the **Project** menu is enabled and labeled with the name of the build step pattern that will be used to perform the build step. (If the project template does not define a build step for the selected file type, the menu item and the button are disabled.)

The action(s) associated with the build step are executed and the results are displayed in the Build Log pane.



## Force building a project

Select **Force Build** if you want to build all of the files in your project for the selected variant, regardless of how recently they have been built. **Force Build** builds sub-projects if the option is check-marked in the APM Preferences Dialog.

## Building variants

You can also choose to build all variants of your project. To do this, select **Build Variants** from the **Project** menu.

## 2.2.4 Build steps

A build step is a step in the build process that contributes to the project output. Usually, a build step generates one file or a group of related files. The action(s) performed in a build step are defined by a build step pattern within the project's template. The build step pattern also defines what type of file will be acted on by each build step. Typical build steps include compiling, assembling, linking or building of sub-project.

## 2.2.5 Project output

A project's output is typically a single file or a closely related group of files, for example a program image, an object library or a sharable library and its "stubs". A project's output is determined by its underlying project template, specifically by one or more of the template's build step patterns. A different version of the project output is created for each variant built (for example, a debuggable version and a releasable version).

## 2.2.6 Project hierarchy

A typical project hierarchy defines the structure of your project in the following sequence:

### **Project**

(my\_proj)

### **Variant**

(eg. Release or Debug)

### **Partition**

(eg. Target, Dependencies, Source)

### **Source**

(eg. sub-proj.apj, my\_source.c,  
my\_source.o, sub-proj.o)

When you build a project using different tool configurations or variable values, variant settings take precedence over project settings, partition settings take precedence over variant settings and source settings take precedence over partition settings.

# ARM Project Manager

---

## 2.2.7 Variants

You can create different versions of your project output(s) from the same source files using variants. Typically you would use variants to create a debug version and a release version of your project output. By changing variant level variables you can control how the project output for the variant is built.

The derived files (eg. object files, the project output) created by APM when you build a project are created in a subdirectory (eg. `Debug`) of the project directory (the project root directory is specified when you create the project).

**Note** *For this release of APM it is not possible to add a source to only one variant of your project.*

## 2.2.8 Partitions

A partition is similar to a directory structure that you might use to separate the various files that make up your project. Partitions help to control the effect of adding a file to a project. The partitions created for a project are determined by the project template. The partitions used by standard APM templates include:

<b>Image/Library</b>	Contains the project output of your build as specified in the project template.
<b>Libraries</b>	Contains any libraries that are to be used by the project.
<b>IncludedFiles</b>	Contains any files included by the sources used by the project.
<b>SubProjects</b>	Contains other projects that are to be used in the construction of the project output. If the project output from a sub-project is a library, the library file is built in the Libraries partition.
<b>Objects</b>	Contains the object files built from the sources.
<b>Sources</b>	Contains source files used to build the project output. Other source partitions that may be added depending on the template, such as Thumb-C, ARM-C, ASM-Sources.
<b>Misc</b>	Anything else you want to add to a project.

**Note** *There is no restriction on the names that you can use for partitions, this list is only an example.*

Files are associated with a partition by APM. A partition is only a construct of APM and files are not physically moved to a partition when they are added to a project.

## 2.2.9 Project templates

A project template defines how a particular type of project output will be built. A template uses build step patterns to describe how various tools work together to construct the project output from a set of source files.

Standard templates are provided with APM, and can be modified as needed to meet your projects' requirements.



There are two incarnations of a template. The first is the empty template described in **2.9.2 Project templates supplied with APM** on page 2-34. Empty template files contain the necessary project configuration information needed to create a particular type of project, such as a Thumb executable image.

The second incarnation is when a template becomes a part of a project; an empty template is turned into a project by creating a new project based on the template, then adding files to it.

In this chapter, editing a template or any of its elements (details, variables, paths, build steps) implies editing a template associated with a project, not editing one of the standard templates. **2.9.6 Creating a new template** on page 2-39 discusses creating and editing new templates.

## 2.2.10 Build step patterns

A build step pattern:

- associates a tool or tools (eg. armcc or armlink) with a build step within a project template
- defines the inputs and outputs associated with a build step
- associates the file types conventionally used and generated by a tool with the partitions used to organize the project
- defines the command line(s) to be used by the tool(s) when the build step is executed during the building of the project

## 2.2.11 Variables

A variable holds a value used by either APM or by a build step pattern to specify a changeable aspect of your project, such as a file name or directory path. A variable prefixed with '\$' is read-only, a variable prefixed with '\$\$' has side effects on the actions of APM. A variable containing a '\$' (eg. path\$Debug) has a standard purpose defined by APM.

Variables can be set for any level of the project hierarchy.

The standard variables are:

\$\$DepthOfDotAPJBelowProjectRoot	Affects how a file's location will be resolved in the project's directory structure.
\$\$ProjectName	When a project is created, this variable is set to " " and the name of the project is contained in \$projectname. When you change the value of this variable, \$projectname is set to the new value.
\$projectname	Contains the project's name as assigned when a project is first created or whenever it is saved. This value is also changed if the variable \$\$ProjectName

# ARM Project Manager

---

<code>path\$variant</code>	is set to a non-empty value. (read only) The path used to specify the variant's directory, created as a sub-directory of the directory where the project file is located. This value can be changed if necessary.
<code>config\$tool</code>	These variables store encodings of a tool's configuration or command-line arguments for tools which are not configurable.

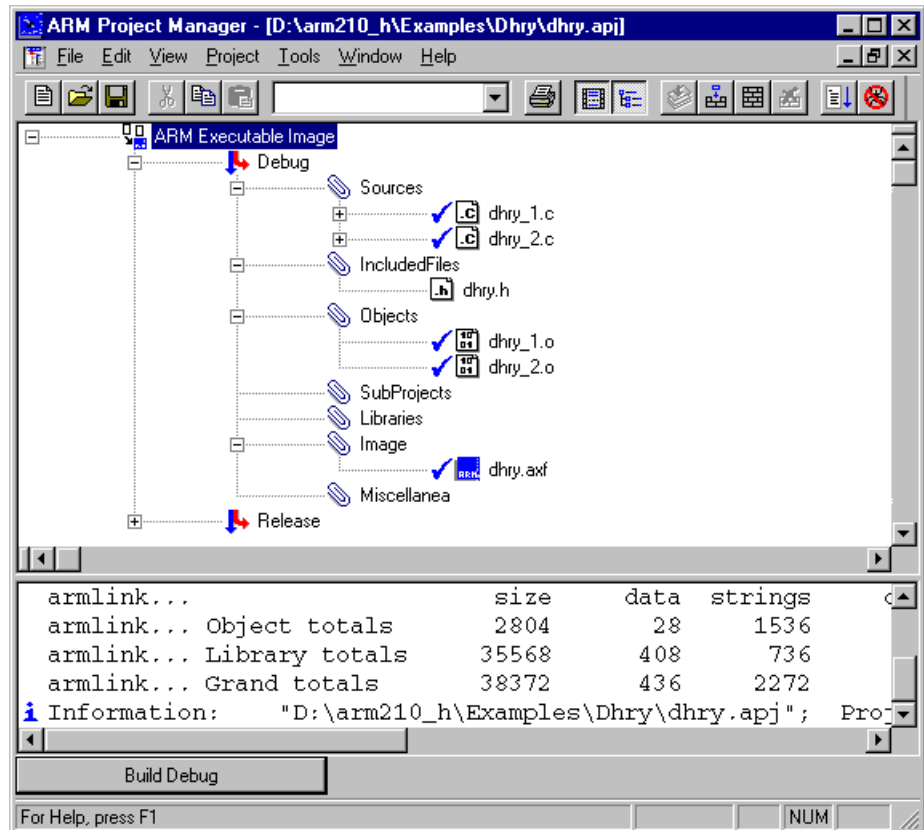
**Note**     *User-defined variables cannot begin with '\$'*



## 2.3 The APM Desktop

### 2.3.1 Project Window

The Project Window consists of a menu, a toolbar, and three panes; the project view, the build log and the status area.



#### Project view

The project view is the topmost pane and displays the project hierarchy. The following symbols are used to denote variants and partitions; file symbols are defined on page 2-3:



variant

partition

# ARM Project Manager

Using the project view, you can examine various aspects of your project and select elements for action. For example you can select a partition to add a file to, or select a source file to be built.

## Build log

The build log is the bottommost pane and is automatically displayed each time you perform a build step or build the entire project. The build log contains messages from the tools used to build your project and you can double-click on many of them and be taken to the line where the error was detected. The following symbols are used in the build log to indicate error severity:

	Informational (blue)
	Warning (blue)
	Error (red)
	Serious Error (red)
	Fatal Error (black)

## Status area

The bottom area of the Project Window contains a button for starting or stopping a build and a progress display. The status bar displays current status information or describes the user interface component (such as a menu option) currently selected.

### 2.3.2 Changing the way a project is displayed

You can change the level of detail displayed in the Project Window in several ways.



By clicking the **View Dependencies** button, you can show or hide lower level the dependency files, which are indicated by a plus sign (+) next to the source file name.



By clicking the **View Build Log** button, you can toggle display of the build log, which is the pane at the bottom of the APM Window. (The build log is automatically displayed when you build a source or an entire project.)

Hold the Tab key and click the mouse on a point in the project view where you want to set a tab to control the spacing of the project hierarchy. All levels of the hierarchy are spaced evenly based on the position you have selected.

You can select **Variants** from the **View** menu to toggle display of the project's variants (partitions and their files are still displayed).

You can also select **ToolBar** or **Status Bar** from the **View** menu to toggle display of the toolbar and status bar.

**Note** *There is currently no expand all feature.*

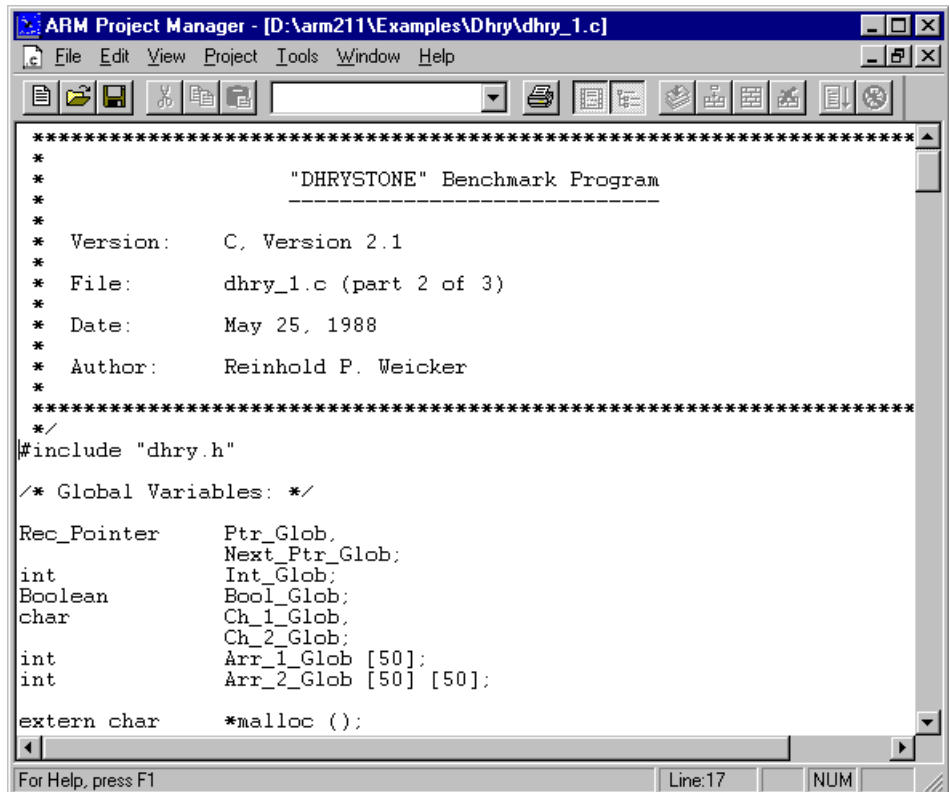


## 2.3.3 Edit Window

You use the Edit Window to create or modify a source file, such as a code file or an include file. This window is opened automatically when you:

- double-click on a code or include file in the Project View
- open an existing code or include file
- select **New** from the **File** menu and create a source or include file

The Edit Window provides a fully functional editor in which you can copy, paste, search and replace using the appropriate toolbar buttons or the **Edit** menu.



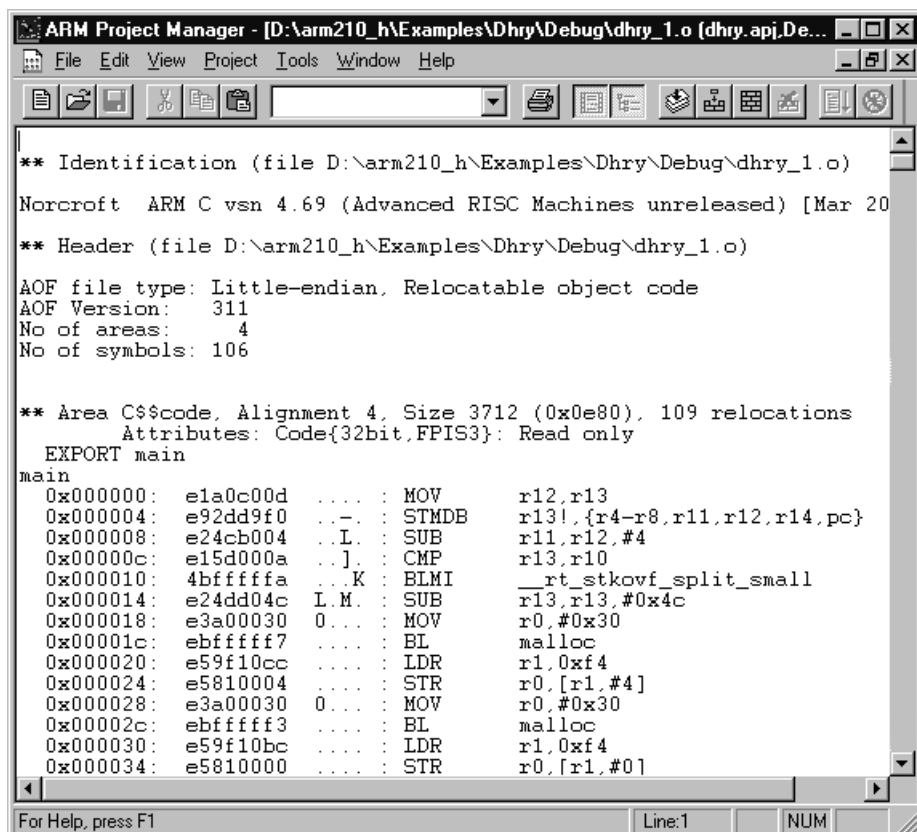
If you are editing a source file as a component of a project, you can build the source file from the Edit Window by clicking the **Perform Build Step** button. You are asked to save the file if you have not already done so, the Edit Window is closed, the Project Window is displayed and the results of the build are displayed in the build log.

**Note** *To build a source file, you must access it as a part of a project because APM uses the project template to determine the way the file is to be built.*

# ARM Project Manager

## 2.3.4 View Window

The View Window is used to display the contents of a binary file (eg. object, library or image file) using a translator such as decaof, decaxf or armlib.



The screenshot shows the ARM Project Manager application window. The title bar reads "ARM Project Manager - [D:\arm210\_h\Examples\Dhry\Debug\dhry\_1.o (dhry.apj,De...". The menu bar includes File, Edit, View, Project, Tools, Window, and Help. The toolbar contains various icons for file operations and viewing. The main text area displays the following information:

```
** Identification (file D:\arm210_h\Examples\Dhry\Debug\dhry_1.o)
Norcroft ARM C vsn 4.69 (Advanced RISC Machines unreleased) [Mar 20
** Header (file D:\arm210_h\Examples\Dhry\Debug\dhry_1.o)
AOF file type: Little-endian, Relocatable object code
AOF Version: 311
No of areas: 4
No of symbols: 106

** Area C$$code, Alignment 4, Size 3712 (0x0e80), 109 relocations
Attributes: Code{32bit,FPIS3}: Read only
EXPORT main
main
0x000000: e1a0c00d .... : MOV r12,r13
0x000004: e92dd9f0 ...- : STMDB r13!,{r4-r8,r11,r12,r14,pc}
0x000008: e24cb004 ..L. : SUB r11,r12,#4
0x00000c: e15d000a ...] : CMP r13,r10
0x000010: 4bfffffa ...K : BLMI __rt_stkovf_split_small
0x000014: e24dd04c L.M. : SUB r13,r13,#0x4c
0x000018: e3a00030 0... : MOV r0,#0x30
0x00001c: ebfffff7 .... : BL malloc
0x000020: e59f10cc .... : LDR r1,0xf4
0x000024: e5810004 .... : STR r0,[r1,#4]
0x000028: e3a00030 0... : MOV r0,#0x30
0x00002c: ebfffff3 .... : BL malloc
0x000030: e59f10bc .... : LDR r1,0xf4
0x000034: e5810000 .... : STR r0,[r1,#0]
```

At the bottom of the window, there is a status bar with the text "For Help, press F1", a "Line:1" indicator, and a "NUM" button.



## 2.4 Getting Started

This section works through the creation of a simple project introducing:

- starting APM
- creating a new project
- adding files to a project
- viewing a project
- building a project
- correcting problems
- using the project output
- exiting APM

### 2.4.1 Starting the APM



If you are running Windows 95, select ARM Project Manager from the ARM SDT V2.11 Program folder.

If you are running Windows NT, double-click on the ARM Project Manager icon in the ARM SDT V2.11 Program group.

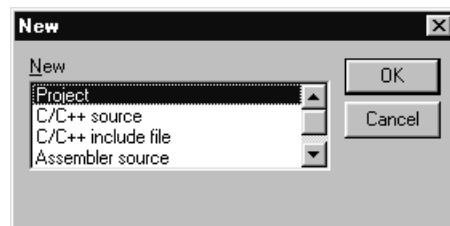
When APM is started, the last file or project accessed will be loaded.

### 2.4.2 Creating a new project

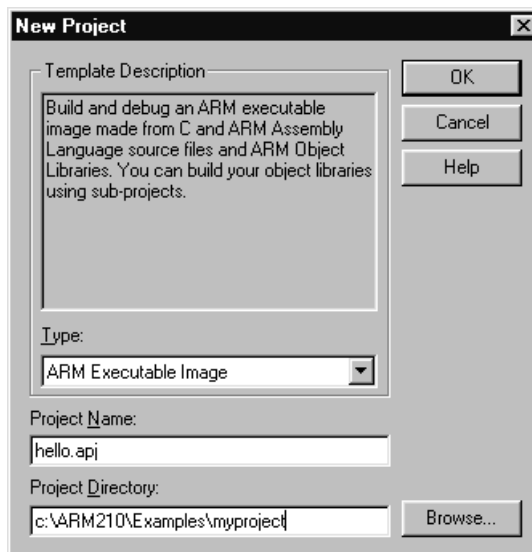
To create a project you perform the following steps:



- 1 Select **New** from the **File** menu or click the **New** button. The New dialog is displayed.



- 2 Select **Project** from the scroll box.
- 3 Click **OK**. The New Project dialog is displayed.



- 4 Select the template **Type** that you want to use for the project. The template description is displayed. For this example select **ARM Executable Image**.
- 5 Enter a **Project** name, such as `hello`. This is used for the project file and the project output.
- 6 Modify the **Project Directory** to `ARM210\Examples\hello`. When you build the project, the variant directories are created within this directory.  
(If the directory doesn't exist, you are prompted to confirm its creation.)
- 7 Click **OK**.

The new project file is created in the project directory and the Project Window is displayed.

## 2.4.3 Creating a new source file

For the `hello` project you created in the previous section, you will create a new source file from within the Project Manager. To create a new file, perform the following steps:

- 1 Click the **New** button or select **New** from the **File** menu. The New dialog appears.
- 2 Select C/C++ source from the scroll box.
- 3 Click **OK**. An Edit Window is displayed.
- 4 Enter the following code:

```
#include <stdio.h>
int main(void)
{
    printf("Hello World")
}
```

```
return 0 ;
```

```
}
```



- 5 Save the text as `hello.c` by selecting **Save** from the **File** menu or clicking the **Save** button.
- 6 Close the Edit Window by selecting **Close** from the **File** menu. You are now returned to the Project Window with the `hello` project loaded.

## 2.4.4 Adding files to a project

To add the newly created file to your project, perform the following steps:

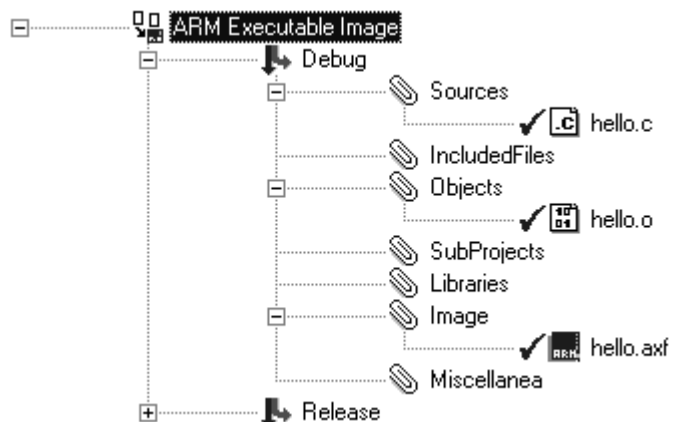
- 1 Ensure the `hello` project is selected as the current project.
- 2 Select **Add file to project** from the **Project** menu.
- 3 Select `hello.c`, navigating to the correct directory if necessary.
- 4 Click **OK**.

The file has now been added to the project.

**Note** *If you haven't changed directories since you created the project, the source file will be created in the same directory as the project. However, you don't have to put source files in the project directory—you can add source files to a project from any directory on your disk.*

## 2.4.5 Viewing the project

Once you have added files to your project, you may want to view the project in more detail. To expand a level of the project hierarchy, click on the plus (+) symbol next to that level. If you expand the first three levels of `hello.apj`, the Project View looks like this:



# ARM Project Manager

You can see that `hello.c` (the file you added to the project) is now in the **Sources** partition. The files `hello.o` and `hello.axf` have been added to the **Objects** and **Image** partitions. These are the derived files that are the anticipated output of the project's build steps. The work in progress symbol indicates that they are not yet complete.



Other options for changing the way a project is displayed are discussed in **2.3.2 Changing the way a project is displayed** on page 2-10.

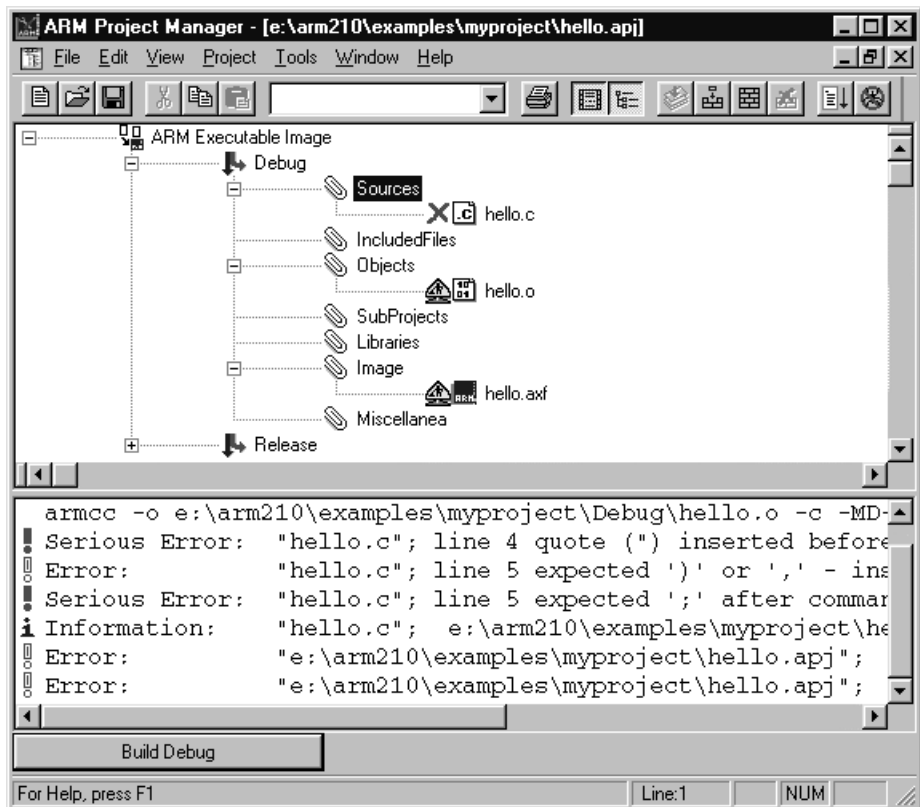
## 2.4.6 Building the project




Now that your project contains source files, it can be built. The simplest way to do this is to click the **Build** button.

When you initiate the build, the button in the status area changes to Stop Build, a build status indicator appears next to this button. Also, messages from the build tools are displayed in the build log, which is automatically displayed if it is not already open.

When the build is complete, the APM desktop should look like this:




## 2.4.7 Correcting problems

-  You can see from the messages in the Build Log (displayed in the previous section) that the build was not successful. The red X by `hello.c` in the project view indicates the source of the problem. You will see in the next section how to use APM to help you resolve problems in building your project.

When you have built your project, there may of course be errors and problems. As the build progresses, messages are written to the build log (see **2.3.1 Project Window** on page 2-9) which is displayed in the build log pane at the bottom of the Project Window. These may be informational messages or diagnostic messages from the tools (eg. compilers or linkers) associated with the project template.

When the build is complete, you can double-click on any error message that relates to an editable source file (eg. compiler errors with file line tags) and APM takes you to the location where the error was detected. In the case of a compile error, this is the line of code listed in the log. If a line pertains to a sub-project, the project is loaded into the Project Window. You can also locate errors using **Next Error** and **Previous Error** on the **View** menu.

To find and correct the problem, then rebuild the project:

- 
- 1 Double-click on the serious error line, indicated by a solid red exclamation mark, in the build log. You are taken to the line in `hello.c` where the error was detected.
  - 2 Correct the problem (a missing semicolon) in the previous line:  

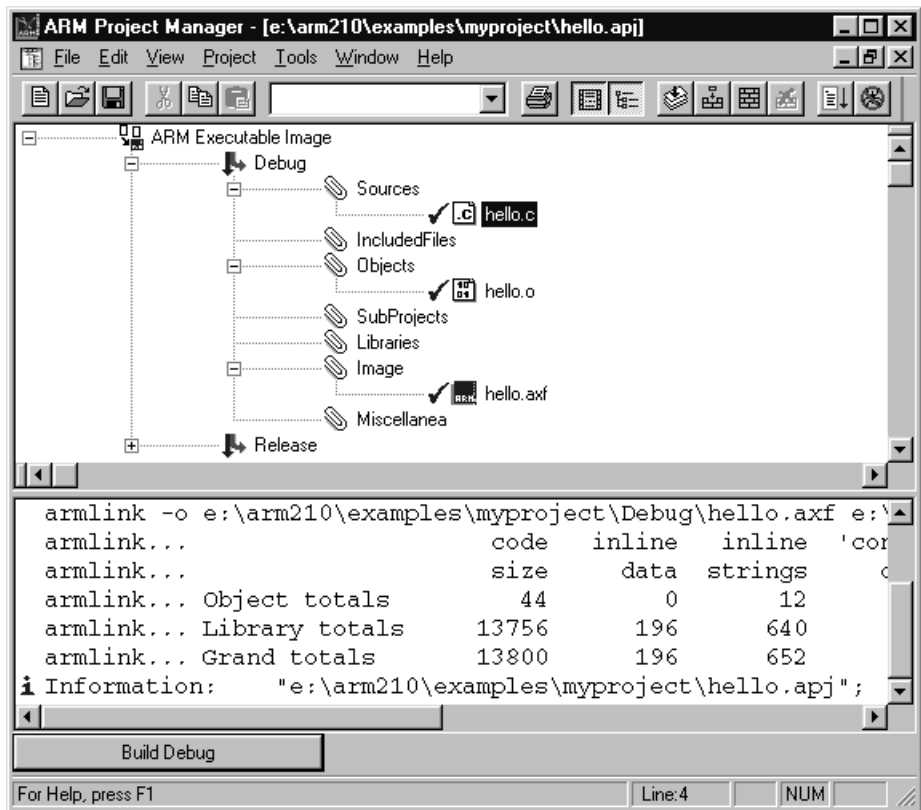
```
printf ("Hello World")
```

should read

```
printf ("Hello World") ;
```
  - 3 Click the **Build** button to rebuild the project. The following occurs:
    - APM prompts you to save the file if you have not already done so
    - the Project Window becomes the current window
    - the build commences and messages are written to the build log

# ARM Project Manager

When the build completes, the APM Desktop should look like this:



- ✓ There are blue checkmarks next to all three files and an informational message in the build log saying the project is up to date.

You can now go on to execute or debug your project.

## 2.4.8 Using the project output

Once your project has been successfully built, you can either execute or debug it to see how it works. Using the ARM Debugger for Windows (see **Chapter 3, ARM Debugger for Windows** for more information).

### Executing an image



When you click the **Execute** button or select **Execute project.apj** from the **Project** menu, the image is loaded in to the ARM Debugger for Windows and processing commences. For hello.apj, "Hello World" appears in the Console Window.

## Debugging an image



When you click the **Debug** button or select **Debug project.apjf** from the **Project** menu, again the image is loaded into the ARM Debugger for Windows, but debugger is halted at the start of the program. You can then debug the image as necessary, using all of the features of the Debugger.

**Note** *Whether you Execute or Debug your image, if the project output is older than its source, it will be rebuilt before it is sent to the Debugger.*

## 2.4.9 Exiting the Project Manager

To exit the Project Manager select **Exit** from the **File** menu. The source files and projects you currently have open will be re-opened the next time you start the Project Manager.

# ARM Project Manager

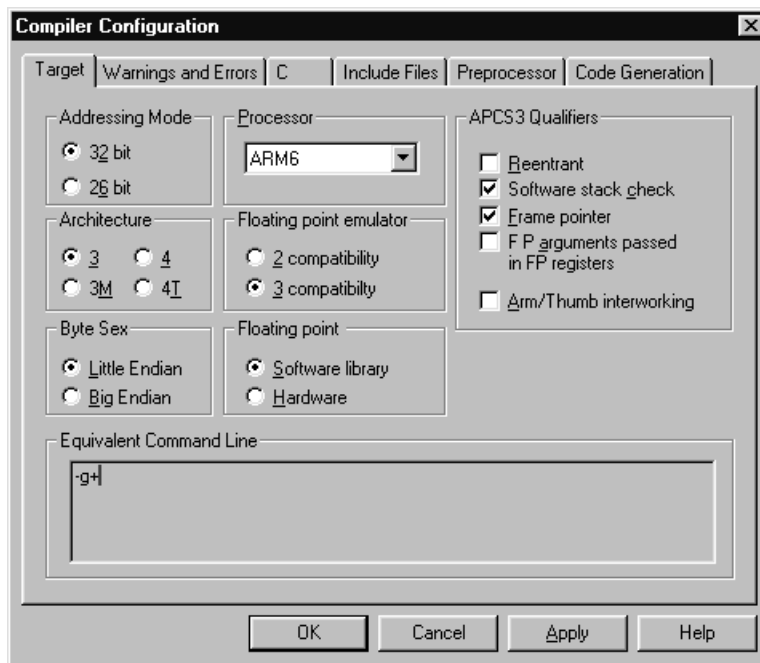
## 2.5 Additional APM Functions

### 2.5.1 Configuring tools

You can change how a tool (eg. compiler or assembler) is executed by changing its configuration within the Project Manager. You can change either the system-wide configuration, or project specific configuration—see page 2-21.

#### The Tool Configuration dialog

The appearance of the Tool Configuration dialog depends on the tool being configured. Tools that are APM compliant (eg. armcc, tcc, armasm, tasm, armlink) respond by displaying their configuration interface. For most ARM tools this consists of sets of property sheets from a .DLL associated with the tool. The following example is for the ARM C compiler:



As you change the settings, the modifications are reflected in the Equivalent Command Line box at the bottom of the dialog.

If APM can't find the necessary set of entry points, either because the tool doesn't have one or because APM failed to locate the tool .DLL (eg. the tool was not installed properly), an error will occur. You must add the tool using a build step pattern and change any settings using the command-line box in the Failed to Locate the Tool dialog.



## Making system-wide configuration changes

System-wide configuration changes affect the tool's use in every project where it is used.

- 1 Select **Configure** from the **Tools** menu.
- 2 Select the tool to configure. The Tool Configuration dialog is displayed.
- 3 Change the configuration as necessary.
- 4 Exit the dialog by:
  - clicking **OK** to save the changes and close the dialog
  - clicking **Apply** to save the changes made
  - clicking **Cancel** to ignore all changes not applied and close the dialog

## Making project-specific configuration changes

Project specific configuration can effect an entire project or specific entities or scope. For example, you could change how a particular source file is compiled, or you could change how all of the source for a specific partition is compiled. The **Project** menu item **Tool Configuration for** reflects the scope that will be impacted by the change.

- 1 Click on the entity (eg. click on the Debug variant in the Project Window) to select the scope for the configuration change.
- 2 Select **Tool Configuration for** from the **Project** menu, then select the tool to be configured. The Tool Configuration sub-menu is displayed.
- 3 Click **Set**. The Tool Configuration dialog is displayed.
- 4 Change the configuration as necessary.
- 5 Exit the dialog by:
  - clicking **OK** to save the changes and close the dialog
  - clicking **Apply** to save the changes made
  - clicking **Cancel** to ignore all changes not applied and close the dialog

## Resetting a tool's configuration

You can return a tool's configuration to the system wide settings (those set using **Set Configuration** from the **Tools** menu) by performing the following:

- 1 Click on the entity (eg. click on the Debug variant in the Project Window) to select the scope for the configuration change.
- 2 Select **Tool Configuration for** from the **Project** menu, then select the tool to be configured. The **Tool Configuration** sub-menu is displayed. If there is a configuration setting for the selected scope, the **Unset** menu item is enabled.
- 3 Select **Unset** and the configuration is reset to the tool's system-wide configuration settings.

# ARM Project Manager

---

## 2.5.2 Force building a project



You can force the building of all your project's sources. All sources, regardless of when they were last built, are rebuilt.

To force build a project:

- select **Force Build *projectname.apj* "variant"** from the Project menu

OR

- click the **Force Build** button

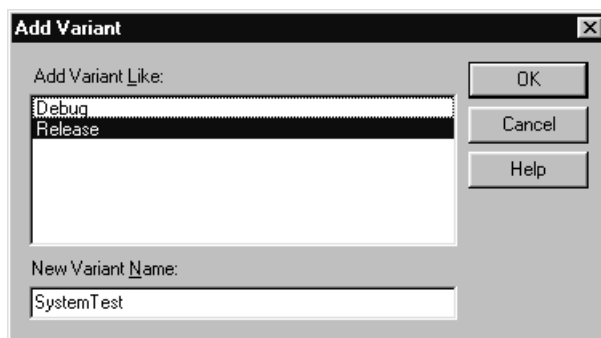
**Note** *If you move your project to a new location, you must rebuild it using **Force Build**.*

## 2.5.3 Adding a new variant to a project

Using variants you can create different versions of your project output(s) from the same source files. Typically you would use variants to create a debug version and a release version of your project output. By changing variant level variables you can control how the project output for the variant is built.

To add a new variant to your project template:

- 1 Select **Add Variant** from the **Project** menu. The Add Variant dialog is displayed.



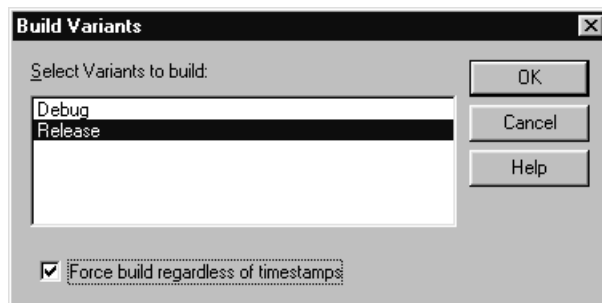
- 2 Select a variant from the **Add Variant Like** list. The files and variable values from the original will be assigned to the new template.
- 3 Enter a **New Variant Name**. The variant name cannot contain spaces.
- 4 Click **OK**.

**Note** *For this release of APM it is not possible to add a source to only one variant of your project.*

## 2.5.4 Building selected variants

To build a variant or variants:

- 1 Select **Build Variants** from the Project menu. The Build Variants dialog is displayed.



- 2 Select one or more variants from the **Select Variants to build** box.
- 3 If you want to force build the selected variants, check **Force build regardless of timestamps**.
- 4 Click **OK** to initiate the build.

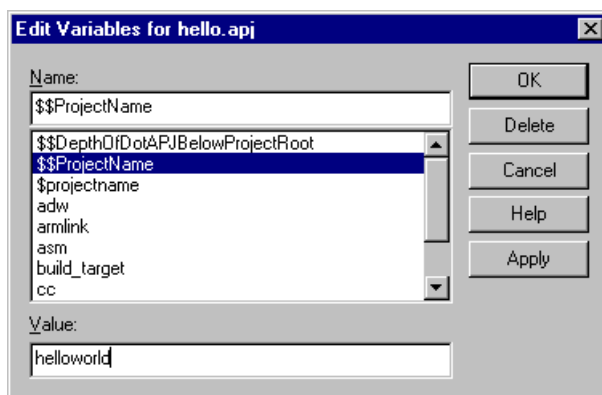
# ARM Project Manager

## 2.5.5 Changing a project's name

There are two options for changing a project's name—changing the name of project output only and changing the project file and the project output. In both cases the original files remain.

### To change the name of the project output:

- 1 Select **Edit Variables for *projectname.apj*** from the **Project** menu. The Edit Variables dialog is displayed.



- 2 Select the variable `$$ProjectName`, enter the new name in the **Value** text box and click the **OK** button.

### To change the name of both the project file and the project output:

- 1 Select **Save As** from the **File** menu. The Save As dialog is displayed.
- 2 Save the file with the new name.
- 3 Rebuild the project.

**Note** *The value of `$$ProjectName` must be " " or the project output will retain the name stored in that variable.*

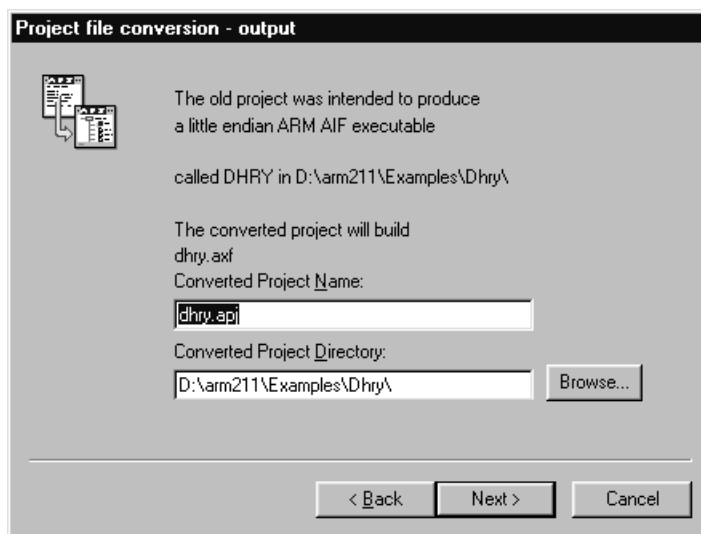
The value of `$projectname` will be updated by APM. See **2.2.11 Variables** on page 2-7 for more information on variables.

## 2.5.6 Converting old projects

If you have a project created with the previous version of the ARM Project Manager, you will be asked to confirm the project's conversion to the new format. Once a project file has been converted, it cannot be read by APM v1.0.

When you open a project in APM v1.0 format:

- 1 The Project Conversion Wizard is loaded.
- 2 Confirm that the conversion should proceed, and that the old file can be overwritten.
- 3 Click the **Next** button to proceed.



- 4 Verify the project name and directory, then click the **Next** button.
- 5 Verify the source files that are to be added to the project, and click the **Next** button. By default all files belonging to the original project will be carried over to the new project file.
- 6 Confirm that you want to proceed with the conversion by clicking the **Finish** button. If you have elected to overwrite the existing file, the conversion cannot be reversed.

## 2.5.7 Stopping a build



You can stop the build after the current step by clicking the **Stop build** toolbar button or by clicking the **Stop Build variant** button in the status area at the bottom of the Project Window.

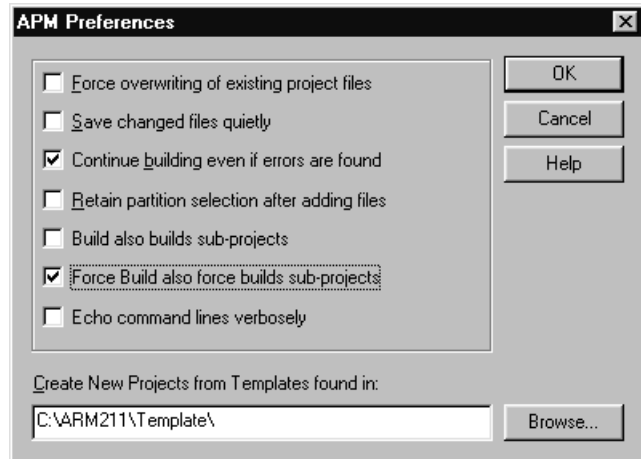
# ARM Project Manager

---

## 2.6 Project Manager Configuration

### 2.6.1 APM preferences

There are several options for creating and building available on the APM Preferences dialog.



#### **Force overwriting of existing project files**

If selected, when you create a new project with an existing file name, the original project file is overwritten without a request for confirmation.

#### **Save changed files quietly**

If selected, changed files are saved without prompting when the project is closed.

#### **Continue building even if errors are found**

If not selected, the build stops on the first error. If set, the build continues ignoring files that depend on erroneous components.

#### **Retain partition selection after adding files**

If selected, the partition a file was last added to retains the focus for the next file addition. This is useful when a file type can be stored in more than one partition.

#### **Build also builds sub-projects**

If selected, all files in all sub-projects will be checked and built as necessary to bring the project up-to-date. If not selected, a sub-project's build target will not be rebuilt, even if it contains files that are out of date.

This setting is useful when the interfaces of library files are stable and the main project's build time should not be impacted by changes to the implementation of the libraries built by the sub-project. The sub-project can be build separately before

building the main project if required.

## **Force Build also force builds sub-projects**

If selected, Force Build builds files in all sub-projects.

## **Echo command lines verbosely**

If selected, command lines that invoke tools will be echoed in full in the build log. This is useful for understanding or auditing project build behavior. Echoing command lines verbosely shows the result of merging tool configurations at the tool, project, variant, partition and file level.

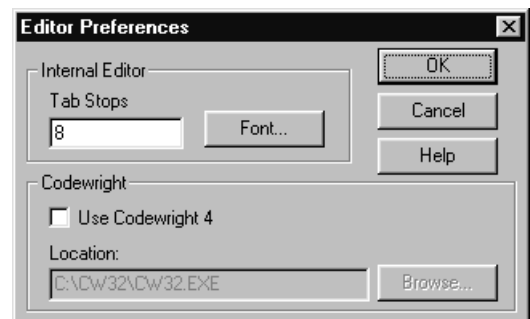
## **Create New Projects from Templates found in:**

Specifies the location of the project template definitions. The default is the `Template` directory off the main ARM installation directory.

An item is check-marked when selected.

## 2.6.2 Editor preferences

You can also select several options to be used when you edit a source or include file, using the Editor Preferences dialog.



### **Internal Editor - Tab Stops**

Changes the tab stops used in the Edit Window.

### **Font**

Displays a standard font dialog.

### **Use CodeWright 4**

Check-mark to select CodeWright 4 (if installed on your machine).

### **Location**

Specify the location of CodeWright if different from the standard installation.

# ARM Project Manager

---

## 2.7 Working with Source Files

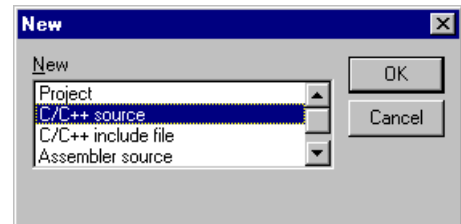
As stated earlier (**2.4.3 Creating a new source file** on page 2-14), you can use APM to edit your source files (ie. .c, .h, .s files). You can use Codewright Version 4 or APM's built-in editor. You select your editor using the Editor Preferences dialog (**2.6.2 Editor preferences** on page 2-27).

If you have selected the file from the Project View, you can build your source file from within the editor. Any messages from the build tool are written to the build log. (You can edit files that are not associated with a project, but until they are added to a project, APM has no information on how the file should be built.)

### 2.7.1 Creating a new source file with APM



- 1 Select **New** from the **File** menu or click the **New** button. The New dialog is displayed.



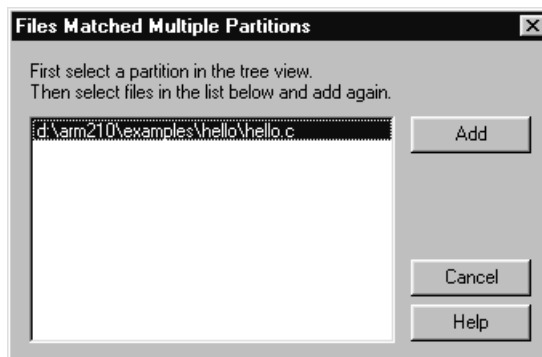
- 2 Select the New file type (eg. **C/C++ source** or **C/C++ include file**) from the scroll box.
- 3 Click **OK**.

**Note** *If you create a new source file and at the same time you have an open project, the source file is not automatically added to the open project.*



## 2.7.2 When a file type is associated with multiple partitions

If the file type has been associated with tools in multiple partitions, the Files Matched Multiple Partitions dialog appears.



This dialog enables you to add the file to the correct partition.

- 1 Select the correct partition in Project Window.
- 2 Select the file or files to add to the selected partition from the Files Matched Multiple Partitions dialog.
- 3 Click **Add**.
- 4 Repeat for each file that is displayed in the dialog.

**Tip** Use the **Retain partition selection settings** on the APM Preferences dialog to keep the default partition set to the one last used.

Project templates supplied with APM have a Misc partition, where file types not associated with other partitions are placed. If for some reason the Misc partition doesn't exist and you add a file not associated with a partition, the Unable to Add Files message box appears. If you still want the file to be added to the project, you must associate the file type with a partition.

If you have a file type that is associated with more than one partition, you can select the partition the file is to be added to before adding it to the project:

- 1 Select the partition from the project view.
- 2 Select **Add Files to partition** from the **Project** menu.
- 3 Select the file or files to be added to the project using the Add Files to partition dialog.
- 4 Click **OK**.

# ARM Project Manager

---

## 2.7.3 Performing a single build step

You can build a single file if it is associated with a project and if the file has been opened as a part of that project. If the file can be built (eg. compiled or assembled) the appropriate menu item in the **Project** menu is enabled and labeled with the name of the build step pattern that will be used to perform the build step. (If the project template does not define a build step for the selected file type, the menu item and the button are disabled.)

For example, if you had several source files in your `hello` project, and selected `hello.c` from the Release variant, the item on the **Project** menu would read:

`Compile hello.c "Release"`

### Note

*"Compile" is a term specified within the template. Any build type tool can be used to build a source file. See **2.10.3 Adding a build step pattern** on page 2-44 for more information on assigning tools to a template.*

To build a single source, perform the following steps:

- 1 Select `hello.c` from the Project Window.



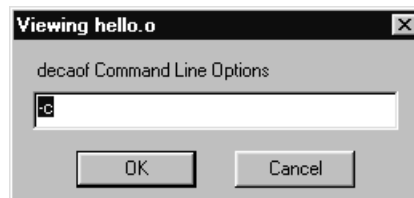
- 2 Click the **Perform Build Step** button (the tool tip reflects which build step will be executed) or select `Compile hello.c "Release"` from the **Project** menu.

The action(s) associated with the Compile build step are executed and the results are displayed in the build log pane.

## 2.8 Viewing Object and Executable Files

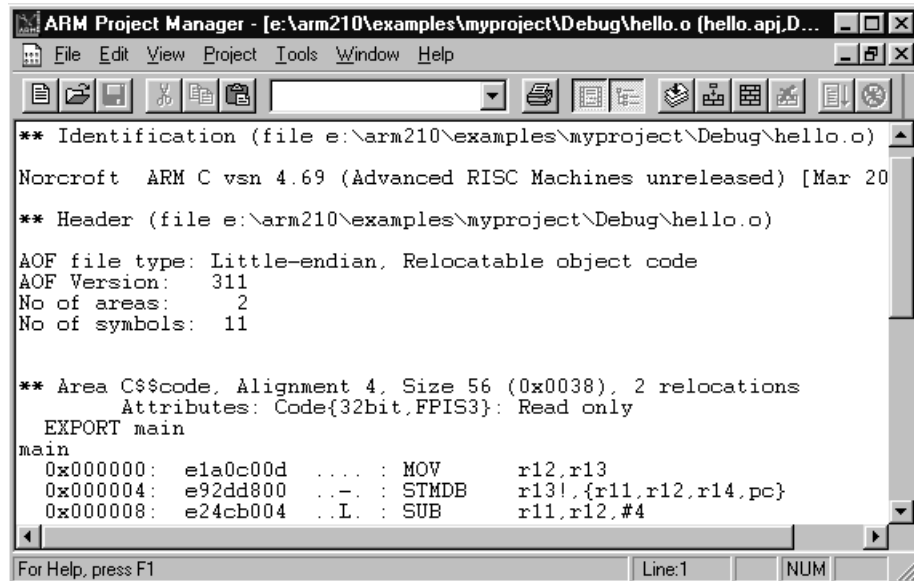
You can view the contents of a binary file (object, library or image) using one of the ARM decoders, decaof, decaxf or armlib from the ARM Project Manager. For example, to display the contents of `hello.o`:

- 1 Select `hello.o` from the Project View.
- 2 Select **Contents *hello.o*** from the **View** menu. The appropriate Viewing dialog for the translator is displayed.



- 3 Use the default command line option, `-c` which displays disassembles code areas (other options are listed in the sections below, or in the on-line help).
- 4 Click **OK**.

The information is displayed in the specified format in a View Window:



# ARM Project Manager

---

## 2.8.1 decaof

The ARM Object Format (AOF) file decoder, decaof, decodes AOF files such as those produced by armasm and armcc. The full specification of AOF can be found in the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

The options that can be specified are:

- a Displays area contents in hex (and implicitly includes -d).
- b Displays on the area declarations (brief).
- c Disassembles code areas (and implicitly includes -d). APM default.
- d Displays area declarations.
- g Displays debug areas formatted readably.
- q Gives a quick report of the area sizes only.
- r Displays relocation directives (and implicitly includes -d).
- s Displays symbol tables.
- t Displays string tables.
- z Displays a one-line code and data size summary per file.

## 2.8.2 decaxf

The ARM Executable Format (AXF) file decoder, decAXF, is a tool that decodes executable files such as those produced by armlink.

The options that can be specified are:

- c Disassembles code areas. APM default.
- g Displays debug areas formatted readably.
- s Displays symbol tables. APM default.
- t Displays string tables. APM default.

## 2.8.3 armlib

armlib options that can be specified are:

- d Displays area declarations.
- r Displays relocation directives (and implicitly includes -d).
- s Displays symbol tables.
- t Displays string tables.

## 2.9 Working with Project Templates

This section goes into detail on what a project template is and how a project template is used within APM.

It also discusses how the various components of a template are modified.

### 2.9.1 General information

#### What is a project template?

Up to this point you have been using the templates supplied with the ARM Project Manager without modification. In this section you will see how to change an existing template in a variety of ways and you will see how you can create templates of your own. The following elements make up a project template:

- build step patterns
- tools
- partitions
- variants
- variables

A build step pattern controls how a specific tool works within the project environment and it specifies how a particular file type is handled within that environment. It controls how a tool transforms its input into output as an intermediate step in building your project's output. A build step pattern has a global effect within a project. However, using variables, you can change that effect at the source, partition or variant level of the project hierarchy. Build step patterns are discussed in section **2.10 Build Step Patterns** on page 2-41.

Tools are the programs used by a build step pattern to transform a source file into a derived file. Tool configuration is discussed in **2.5.1 Configuring tools** on page 2-20.

Partitions are a construct of APM used to organize your project's source and derived files, in the same way that you might use a directory structure. The partitions used by your project are determined by the build step patterns.

Variants are used to create different versions of the project output from the same set of source files, typically a Debug version and a Release version. Partitions are discussed in **2.5.3 Adding a new variant to a project** on page 2-22.

Variables are used within the definition of build steps to change how a tool is used in the various levels of a project hierarchy. For example, you could use a variable to change the configuration settings for armcc, so that a single source, or a particular set of sources in a separate partition would be compiled in one way and the rest of the source is compiled in another see **2.9.3 Editing a variable** on page 2-36.

### Using a template to create a project

When you create a new project, you select a existing template that defines the tools to be used, how the project's files are to be organized into partitions, and the variants that can be built. By adding your source files to a project template, you can easily build your project output.

See **2.4.2 Creating a new project** on page 2-13 for more information.

### Modifying a project template

You can modify a project template after the project has been created or you can create your own master templates.

Once you have created your project you can add new tools, remove unused tools and edit the build step patterns to suit your needs. These changes have a global effect across your project, but do not effect the master template you used when you created your project.

If you need finer control, for example to compile a sub-set of your sources in a particular way, you can use tool configuration settings and/or variables to change the way a single source, or a group of sources are handled. For example, you could create an additional source partition and change the configuration of the tool used on files added to that partition so that they are built differently from the sources in the original source partition or you could even use a different compiler.

Finally, if you know you will create a number of projects that require a framework that is not supported by the supplied master templates, you can create your own masters. To create a master template, simply take an existing project template, save it to a new file in the `ARM\Templates` directory, then make the required changes. The new template will be added to the list presented when you create a new project.

**Note**     *You are advised to create new templates based on the provided APM templates, rather than modifying the master templates supplied with APM.*

## 2.9.2 Project templates supplied with APM

There are several standard templates provided with APM that you can use to create both executable images and ARM libraries:

<b>ARM Executable Image</b>	Build and debug an ARM executable image made from C and ARM Assembly Language source files and ARM Object Libraries. You can build your object libraries using sub-projects.
<b>Thumb Executable Image</b>	Build and debug a Thumb executable image made from C and Thumb/ARM Assembly Language source files and Thumb Object Libraries. You can build your object libraries using sub-projects. You can compile some C sources for ARM state by setting the <code>cc</code> Project variable to <code>armcc</code> for just those source files ( <b>2.9.3 Editing a variable</b> on page 2-36).



<b>ARM Object Library</b>	Build a library of ARM object files from C and ARM Assembly Language source files. You can use the library as a component in Projects to build ARM executable images.
<b>Thumb Object Library</b>	Build a library of Thumb object files from C and Thumb/ARM Assembly Language source files. You can use the library as a component in Projects to build Thumb executable images. You can compile some C sources for ARM state by setting the <code>cc</code> Project variable to <code>armcc</code> for just those source files.
<b>Thumb-ARM Interworking Image</b>	Build and debug a Thumb-ARM interworking image made from: Thumb C, ARM C and Thumb/ARM Assembly Language source files; Thumb Object Libraries and ARM Object Libraries. You can build your object libraries using sub-projects.
<b>Blank Template</b>	A blank template from which to make your own templates. Change the description using the Details dialog ( <b>2.9.7 Editing a project template's details</b> on page 2-40). This template defines Debug and Release variants and the Misc partition.

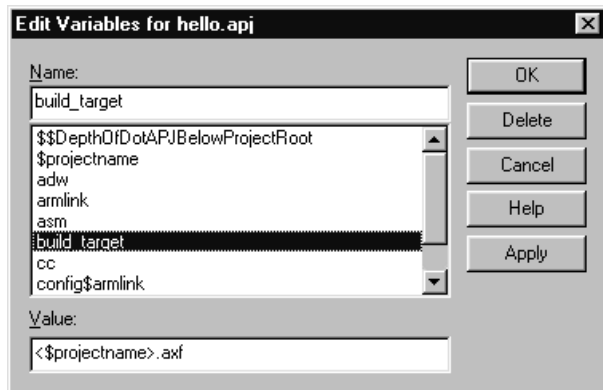
# ARM Project Manager

## 2.9.3 Editing a variable

A variable holds a value used by either APM or by a build step pattern to specify a changeable aspect of your project, such as a file name or directory path. Variables can be set for any level of the project hierarchy. For example, you could set the variable specifying the C compiler (*cc*) to be one tool for the entire project (ie. *cc=armcc*) and create a special configuration for a particular source file (*cc=tcc*).

To edit the variables for a particular level of the project hierarchy:

- 1 Select an element from the Project View (for example the Debug variant).
- 2 Select **Edit Variable** from the **Project** menu. The Edit Variables dialog is displayed, the title of the dialog box reflects the scope of the changes that are going to be made.



- 3 Select a variable from the scroll box or type the variable **Name**.
- 4 Type the new **Value**.
- 5 If you have additional variables to modify, click **Apply** to save the change and modify another variable.
- 6 Once you have completed your changes exit the dialog:
  - Click **OK** to save the changes and exit the dialog.
  - Click **Cancel** to exit the dialog. Changes that have not yet been applied are lost.

The following restrictions apply:

- Variables prefixed by \$ are read-only and cannot be modified or deleted.
- Variables prefixed by \$\$ are reserved for use by APM, but can be modified.
- Variables containing a \$ (eg. *path\$Debug*) have a standard purpose defined by APM.



- Use caution when editing `config$xxx` variables, especially if these contain | symbols. These are the internal representation of tool configurations created by the tools.

## 2.9.4 Editing a path

If you want to use a tool in your project that is not on the Windows' search path:

- 1 Select **Edit Paths** from the **Project** menu. The Edit Paths dialog is displayed:



- 2 Select a tool from the scroll box.
- 3 Change the path as required in the **Edit Path** field.
- 4 If you have additional tool paths to modify, click **Apply** to save the change and go on to modify another path.
- 5 Once you have completed your changes:
  - click **OK** to save the changes and exit the dialog.OR
  - Click **Cancel** to exit the dialog without saving changes that have not yet been Applied.

**Note**     *You don't need to edit paths if your tools are on your Windows' search path.*

# ARM Project Manager

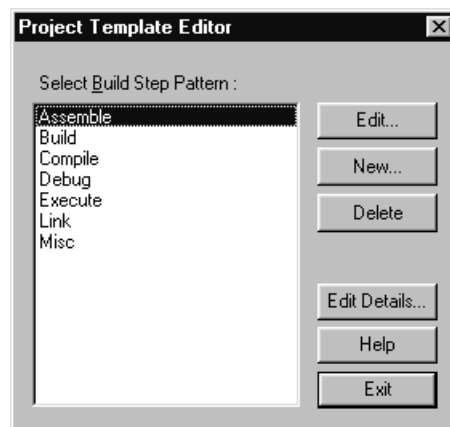
---

## 2.9.5 Editing a project template

When you edit a project template, you change the options used by APM when it builds the project.

To edit a template, perform the following steps:

- 1 Select **Edit Project Template** from the **Project** menu. The Project Template Editor dialog is displayed.



- 2 You can now do one or more of the following:
  - Select a build step pattern and click **Edit** to modify (see **2.10.2 Editing a build step pattern** on page 2-43), or **Delete** to remove.
  - Click **New** to add a new build step pattern (see **2.10.3 Adding a build step pattern** on page 2-44).
  - Click **Edit Details** to change the title and/or description of the template (see **2.9.7 Editing a project template's details** on page 2-40).
- 3 Click **Exit** to close the dialog.

## 2.9.6 Creating a new template

If you have a number of similar project outputs to produce that don't fit the templates provided with the ARM Project Manager, you can create your own project template. You can use an existing template or project as a basis for your new template.

To create a new template:

- 1 Select the foundation for your new project template:
  - Select an existing template by creating a new project (**2.4.2 Creating a new project** on page 2-13).
- OR
- Open an existing project.
- 2 Select **Edit Project Template** from the **Project** menu.
- 3 Click the **Edit Details** button and modify the **Title** and the **Description** of the template—see **2.9.7 Editing a project template's details** on page 2-40.
- 4 Select **Save as Template** from the **File** menu. The Save As dialog is displayed.
- 5 Locate the file in the directory specified in the APM Preferences dialog (**2.6 Project Manager Configuration** on page 2-26) and give it a unique name.
- 6 Click **Save**. The new project has now been created and is the currently active project.
- 7 Modify the build step patterns listed for the template, adding or deleting build step patterns as necessary—see **2.10.2 Editing a build step pattern** on page 2-43.
- 8 Edit any variables as necessary—see **2.9.3 Editing a variable** on page 2-36.
- 9 Edit any tool paths as necessary—see **2.9.4 Editing a path** on page 2-37.
- 10 Save the project.

The new template is displayed (sorted by filename) in the **Type** list of the **New Project** dialog the next time you create a new project.

# ARM Project Manager

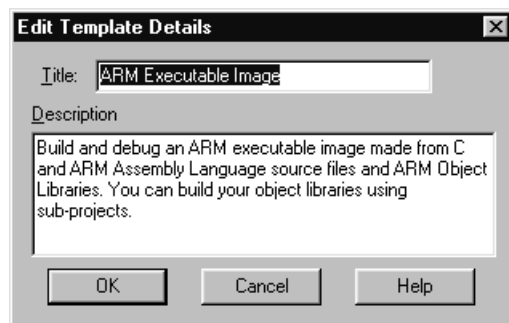
---

## 2.9.7 Editing a project template's details

The template details consist of a short name of the project (ie. ARM Executable Image) and a description providing more details on the project.

To edit a template's details:

- 1 Select **Edit Project Template** from the **Project** menu. The Project Template Editor is displayed.
- 2 Click **Edit Details**. The Edit Template Details dialog is displayed.



- 3 Change the **Title** and/or **Description** as needed.
- 4 Click **OK**.

## 2.10 Build Step Patterns

### 2.10.1 Specifying input and output patterns in a build step pattern

A build step pattern uses simple pattern expressions to describe:

- the inputs to which it can be applied
- the outputs it will generate
- the command line(s) that will be used to generate those outputs

When a file is added to a project, APM searches for an input pattern expression to match the file's extension. If a match is found, the pattern variables used in the input pattern become defined and are used to generate output file names. When the project is built, the same pattern variables are used to generate command lines for the tools invoked by the build step pattern.

#### Input pattern expressions

An input pattern can contain three kinds of pattern element:

Variables      written as `<name>`, that match any sequence of characters not containing the next literal.

Literals        written as is, that match only themselves.

Conditional literals

written as `<name | literal>` that either:

- matches and `<name>` takes on the value of the literal.
- fails to match and `<name>` takes on the value `' '` (null).

Patterns match from right to left, and `'/'` in a pattern matches `'/'` or `'\'` in the file name.

For example:

```
<path><slash|/><file>.c
```

would match `myfile.c` with:

```
path = '  
slash = '  
file = 'myfile'
```

It would also match:

```
c:\projdir\myproj\myfile.c
```

with

```
path = 'c:\projdir\myproj'  
slash = '  
file = 'myfile'
```

**Note**      `<path>\<file>.c` *does not match* `myfile.c` (there is no match of `'\'`).

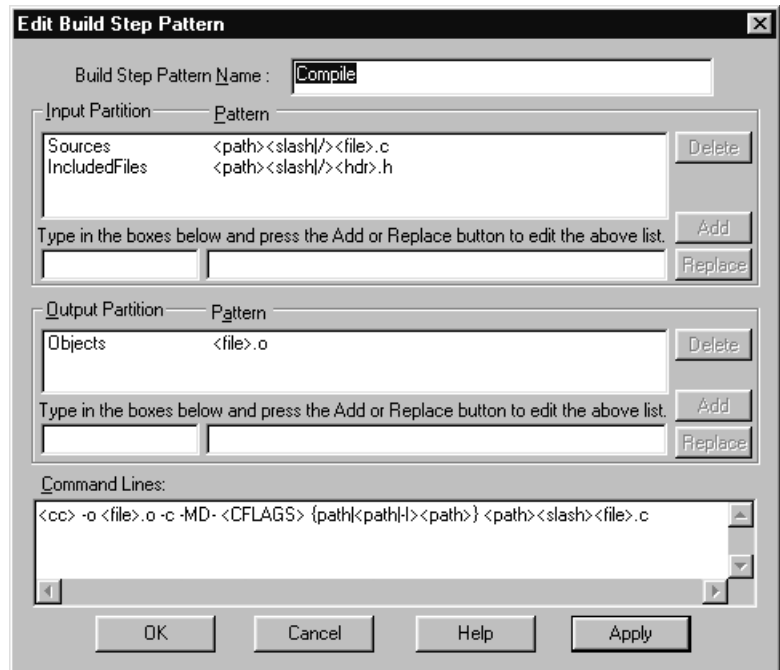


## 2.10.2 Editing a build step pattern

**Note** A build step pattern can use one or more tools, process one or more input files, and can produce one or more outputs.

To edit a build step pattern:

- 1 Select the **Build Step Pattern** from the Project Template Editor dialog.
- 2 Click **Edit**. The Build Step Pattern dialog is displayed.



- 3 Edit the build step pattern as required:

### Deleting a input/output pattern:

- a) Select the pattern line from the list in the **Input** or **Output Partition** box. The selected pattern is loaded on the edit line.
- b) Click **Delete** to remove the pattern.

### Editing a input/output pattern:

- a) Select the pattern line from the list in the **Input** or **Output Partition** box. The selected pattern is loaded on the edit line.

b) Edit the pattern as required (see **2.10.1 Specifying input and output patterns in a build step pattern** on page 2-41).

c) Click **Replace** to change the pattern.

**Adding a new input/output pattern:**

a) Enter an **Input** or **Output Partition**.

b) Enter the pattern for the partition.

c) Click **Add**.

4 Change the **Command Line** as required.

5 Click **OK** to save the changes and exit the dialog.

**Note** *If you don't click **Add**, **Replace** or **Delete** in step 3, you will be prompted to save or cancel your changes before exiting this dialog.*

**Tip** *You can have more than one Edit Build Step Pattern dialog open at one time, so that you can copy from one build step pattern to another easily using **Ctrl+Insert** (to copy text) and **Shift+Insert** (to paste text).*

## 2.10.3 Adding a build step pattern

To add a new build step pattern:

- 1 Select **Edit Project Template** from the **Project** menu.
- 2 Click **New**.
- 3 Enter the name of the new build step pattern.
- 4 Click **OK**. The Edit Build Step Pattern dialog is displayed.
- 5 Specify **Input** and **Output Partition** information:
  - a) Enter a **Partition** name. If the partition does not exist, it will be created.
  - b) Enter the **Pattern**.
  - c) Click **Add**.
- 6 Enter a command line in the **Command Lines** edit box.
- 7 Click **OK**.



# 3

## ARM Debugger for Windows

3.1	Introduction	3-2
3.2	Debugging an ARM Application	3-3
3.3	Debugging Systems	3-3
3.4	ADW Concepts	3-6
3.5	The ADW Desktop	3-11
3.6	Getting Started	3-21
3.7	Debugger Configuration	3-28
3.8	Displaying Image Information	3-34
3.9	Setting and Editing Complex Breakpoints and Watchpoints	3-39
3.10	Other Debugging Functions	3-42

# ARM Debugger for Windows

---

## 3.1 Introduction

The *ARM Debugger for Windows (ADW)* enables you to debug your ARM targeted image using any of the debugging systems described in **3.3 Debugging Systems** on page 3-3.

ADW works in conjunction with either a hardware (eg. ARM Development Board or EmbeddedICE) or software (eg. ARMulator) target system. You debug your application using a number of windows that give you various views on the application you are debugging.

You can also use the ADW to benchmark your application.

Refer to the documentation supplied with your target board for more information on development boards, EmbeddedICE etc.

See **Chapter 9, Basic Assembly Language Programming** for more information on ARM code.

See **Chapter 5, The ARMulator** for more information on the ARMulator.

This chapter provides:

- a basic introduction to ADW
- an overview of the terminology used in ARM debugging
- an overview of the ADW desktop and windows
- a step by step guide to debugging a simple application
- ways of displaying information while debugging
- an overview of more advanced debugging functions

For detailed instructions on how to use ADW, refer to the comprehensive on-line help. In addition, the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) contains a description of the Windows debugger menu options, toolbar and main windows.

Command line debugging is described in **Chapter 4, Command-Line Development**.

### 3.1.1 On-line help

When you have started ADW, you can use on-line help to find information on the tasks you are performing. You have several options for accessing the Help system:

<b>Contents</b>	Select <b>Contents</b> from the <b>Help</b> menu to display a Table of Contents.
<b>Search</b>	Select <b>Index</b> from the <b>Help</b> menu to display an index of all the help topics.
<b>Help</b>	Click the <b>Help</b> button to get information on the dialog currently on display.
<b>F1</b>	Press the <b>F1</b> key on your keyboard to get help on the currently active window or the dialog currently on display.

## 3.2 Debugging an ARM Application

There are a number of ways in which you can debug an application developed to run on an ARM-based system. This section will help you choose a debugging strategy suitable for your application.

To debug your application you must choose:

- a *debugging system*, which may be:
  - hardware-based on an ARM core
  - software which emulates an ARM core
- a *debugger*, such as ADW or armsd (see **Chapter 4, Command-Line Development**).

## 3.3 Debugging Systems

There are three debugging systems available for applications developed to run on an ARM core:

- the ARMulator
- EmbeddedICE
- Angel Debug Monitor (with or without EmbeddedICE)

These systems are described in the following three sections. For details about Demon, the debug monitor supplied with previous versions of the Toolkit, refer to *Application Note 39: Demon and RDP* (ARM DAI 0039).

### 3.3.1 The ARMulator

The ARMulator is a collection of programs that emulate the instruction sets and architecture of various ARM processors. It is instruction-accurate, meaning that it models the instruction set without regard to the precise timing characteristics of the processor. It can report the number of cycles the hardware would have taken. As a result, the ARMulator is well suited to software development and benchmarking. The ARMulator also supports a full ANSI C library to allow complete C programs to run on the emulated system.

The ARMulator:

- provides an environment for the development of ARM-targeted software on the supported host systems
- allows benchmarking of ARM-targeted software (though its performance will be somewhat slow compared to real hardware)

The ARMulator is transparently connected to armsd to provide a hardware-independent ARM software development environment. Communication takes place via the *Remote Debug Interface (RDI)*.

See **Chapter 5, The ARMulator** for more information.

# ARM Debugger for Windows

---

## 3.3.2 EmbeddedICE

EmbeddedICE is a JTAG based debugging system for ARM processors. EmbeddedICE provides the interface between a debugger and an ARM core embedded within any ASIC. EmbeddedICE provides you with:

- realtime address and data-dependent breakpoints
- single stepping
- full access and control of the ARM core
- full access to the ASIC system—full memory access (read and write)
- full I/O system access (read and write)

EmbeddedICE also allows the embedded microprocessor to access host system peripherals, such as screen display, keyboard input and disk drive storage.

See **Chapter 7, EmbeddedICE** for information about using EmbeddedICE and **3.7.6 EmbeddedICE Configuration** on page 3-33 for information on configuration options.

## 3.3.3 Remote\_A

The Angel Debug Monitor (Remote\_A) is a program that allows rapid development and debugging of applications running on ARM-based hardware. Angel can debug applications running in both ARM- and Thumb-state on target hardware.

A typical Angel system has two main components which communicate via a physical link, such as a serial cable.

The debugger runs on the host computer, giving instructions to Angel and displaying the results obtained from it. The debugger could be armsd, ADW, or another debugging tool that can handle the communications protocol used by Angel.

The Angel Debug Monitor runs alongside the application being debugged on the target platform. There are two versions of Angel:

- a full version for use on development hardware
- a minimal version intended for use on production hardware

Angel uses a debugging protocol called the *Angel Debug Protocol (ADP)*, which supports multiple channels. It is easy to port to different hardware. It requires control over the ARM's exception vectors, but can share these with your applications.

Angel can be used for:

- evaluating existing application software on real hardware (as opposed to hardware emulation)
- developing software applications on development hardware
- bringing into operation new hardware that includes an ARM processor
- porting operating systems to ARM-based hardware

If you use Angel for debugging using an ARM Development Board you can simply use the Angel software supplied on the release CD. If you plan to use Angel to debug your application on your own custom hardware you must port Angel to run on this hardware.

See **Chapter 6, Angel** for more information on Angel.

## 3.3.4 Remote\_D

Remote\_D links the host debugger with a remote target system using the RDP debug protocol.

**Note** *Remote\_D has been superseded by the Angel Debug Monitor (Remote\_A). See **6.13 Notes for Demon Users** on page 6-29 for more information.*

# ARM Debugger for Windows

---

## 3.4 ADW Concepts

This section describes the terminology used throughout this chapter.

### 3.4.1 ARM/Thumb code

The ARM assembly language is 32-bit assembly code which can be compiled to run on an ARM core, eg. ARM8.

The Thumb assembly language is 16-bit assembly code which can be compiled to run on a Thumb-compatible ARM core, eg. ARM7TDMI.

See the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for more information.

### 3.4.2 Backtrace

When your program has halted (ie. at a breakpoint or watchpoint), backtrace information is displayed in the Backtrace Window to give you information about the procedures that are currently active.

The following example shows the backtrace information for a program compiled with debug information and linked with the C library:

```
#DHR_2:Proc_6 line 42
#DHR_1:Proc_1 line 315
#DHR_1:main line 170
PC = 0x0000eb38 (_main + 0x5e0)
PC = 0x0000ae60 (__entry + 0x34)
```

Lines 1–3:

The first line indicates the function you are currently in. The second line indicates the source code line from which this function was called, and similarly the third line indicates the call to the second function.

Lines 4–5:

The final line indicates the entry point made by the C library's call into your program and line 4 shows the position of the call into your program's main procedure.

**Note** *A simple assembler program compiled without debug information and not linked to a C library would show only the PC values.*

### 3.4.3 Breakpoints

A breakpoint is a point in the code where your program will be halted by ADW. Once you have set a breakpoint it will appear as a red marker in the left-hand pane of the window.

There are two types of breakpoints:

- a simple breakpoint that stops at a particular point in your code

- a complex breakpoint that:
  - stops when the program has passed the specified point a number of times.
- AND/OR
  - stops at the specified point only when an expression is true.

You can choose to set a breakpoint at a point in the source or in the disassembled code if it is currently being displayed, with the interleaved source option or the disassembly view. You can also set breakpoints on individual statements on a line, if that line contains more than one statement.

You can set, edit or delete breakpoints in the following windows:

- Execution
- Disassembly
- Source File
- Backtrace
- Breakpoints

## 3.4.4 Debug Agent

A debug agent is the entity that actions the debugger's requests, for example to set breakpoints, or read or write to memory. It is not the program being debugged, or ADW itself. Examples of debug agents include the EmbeddedICE hardware, the ARMulator, or the Angel Debug Monitor.

## 3.4.5 Disassembled code

Disassembled code is the machine code generated by the disassembly process.

You can display disassembled code in the Execution Window or in the Disassembly Window (select **Disassembly** from the **View** menu).

You can also choose the type of disassembled code to display by accessing the **Disassembly mode** sub-menu, from the **Options** menu. ARM code, Thumb code or both can be displayed, depending on your program's image type.

## 3.4.6 High- and low-level symbols

A high-level symbol for a procedure refers to the address of the code generated by the first statement in the procedure, and is denoted by the function name shown in the Function Names Window.

A low-level symbol for a procedure refers to its call address, often the first instruction of the stack frame initialization. You can display a list of the low-level symbols in your program in the Low-level Symbols Window.

To indicate a low-level symbol in a regular expression, precede the symbol with @.

To indicate a high-level symbol precede it with a ^.

# ARM Debugger for Windows

---

Refer to the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for information about predefined low-level symbols.

## 3.4.7 Profiling

Profiling enables the programmer to see where most of the processor time is spent within his code, by sampling the PC at time intervals set by the user. This information is then used to build up a picture of the percentage time spent in each procedure. Using the command-line program `armprof` on the data generated by either `armsd` or `ADW`, the programmer can see where effort can be most effectively spent to make the program more efficient.

**Note** *Currently there is no display of profiling information from within the ADW. You must capture the data using the **Profiling** functions on the **Options** menu, then use the `armprof` command-line tool.*

*Profiling is only supported for ARMulator.*

See **Chapter 8, Benchmarking, Performance Analysis and Profiling** for more information on profiling.

## 3.4.8 Regular expressions

Regular expressions are the notation for specifying and matching strings, similar to arithmetic expressions. A regular expression is either:

- a single extended ASCII character (other than the special characters described below)
- a regular expression modified by one of the special characters

You can include low-level symbols or high-level symbols in a regular expression.

The following special characters modify the meaning of the previous regular expression, and will not work if no such regular expression is given.

- |   |  |
|---|--|
| * | Any number of the proceeding regular expressions (including no occurrences). For example, <code>A*B</code> would match <code>B</code> , <code>AB</code> and <code>AAB</code> .   |
| ? | Either one copy of the proceeding regular expression, or nothing at all. For example, <code>AC?B</code> matches <code>AB</code> and <code>ACB</code> but not <code>ACCB</code> . |
| + | At least one copy of the regular expression. For example, <code>AC+B</code> matches <code>ACB</code> and <code>ACCB</code> , but not <code>AB</code> .                           |

The following special characters are regular expressions in themselves:

- |     |   |
|-----|---|
| \   | Precedes any special character you want to include literally in an expression to form a single regular expression. For example, <code>\*</code> matches a single asterisk (*) and <code>\\</code> matches a single backslash (\). The regular expression <code>\x</code> is equivalent to <code>\x</code> as the character <code>x</code> is not a special character. |
| ( ) | Allows grouping of characters. For example, <code>(02)*</code> matches <code>202202202</code> (as well as nothing at all), and <code>(AC?B)+</code> looks for sequences of <code>AB</code> or <code>ACB</code> , such as <code>ABACBAB</code> .   |



- Exactly one character. This is different to `?` in that the period (`.`) is a regular expression in itself, so `.*` matches all, while `?*` is invalid. Note that `.` does *not* match the end-of-line character.
- [ ] A set of characters, any one of which may appear in the search match. For example, the expression `x[23]` would match strings `x2` and `x3`. The expression `[a-z]` would match all characters between `a` and `z`.

**Note** *Pattern matching is done following the UNIX `regex(5)` format, but without the special symbols, `^` and `$`.*

## 3.4.9 Remote Debugging Interface

The *Remote Debug Interface (RDI)* is a procedural interface between a debugger and the image being debugged, via a debug monitor or controlling debug agent. RDI gives the debugger core a uniform way to communicate with:

- a controlling debug agent or debug monitor linked with the debugger
- a debug agent executing in a separate operating system process
- a debug monitor running on ARM-based hardware accessed via a communication link
- a debug agent controlling an ARM processor via hardware debug support

See the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for more information.

## 3.4.10 Search paths

If you want to view the source for your program's image during the debugging session, the ADW needs to know how to find the files. A search path points to a directory or set of directories that are used to locate files whose location is not referenced absolutely.

If you are developing your program using the ARM Project Manager, the search paths are set up automatically.

If you are using the ARM command-line tools to build your project, you may need to edit the search paths for your image manually, depending on the options you chose when you built it.

If for some reason the files have moved since the image was built, the search paths for these files must be set up in the ADW, using the Search Paths Window (see **Search paths** on page 3-34).

## 3.4.11 Watchpoints

In its simplest form, a watchpoint halts a program when a specified register or variable is changed. The watchpoint will halt the program at the next statement or machine instruction after the one that triggered the watchpoint.

There are two types of watchpoints:

- a simple watchpoint that stops when a specified variable changes

# ARM Debugger for Windows

---

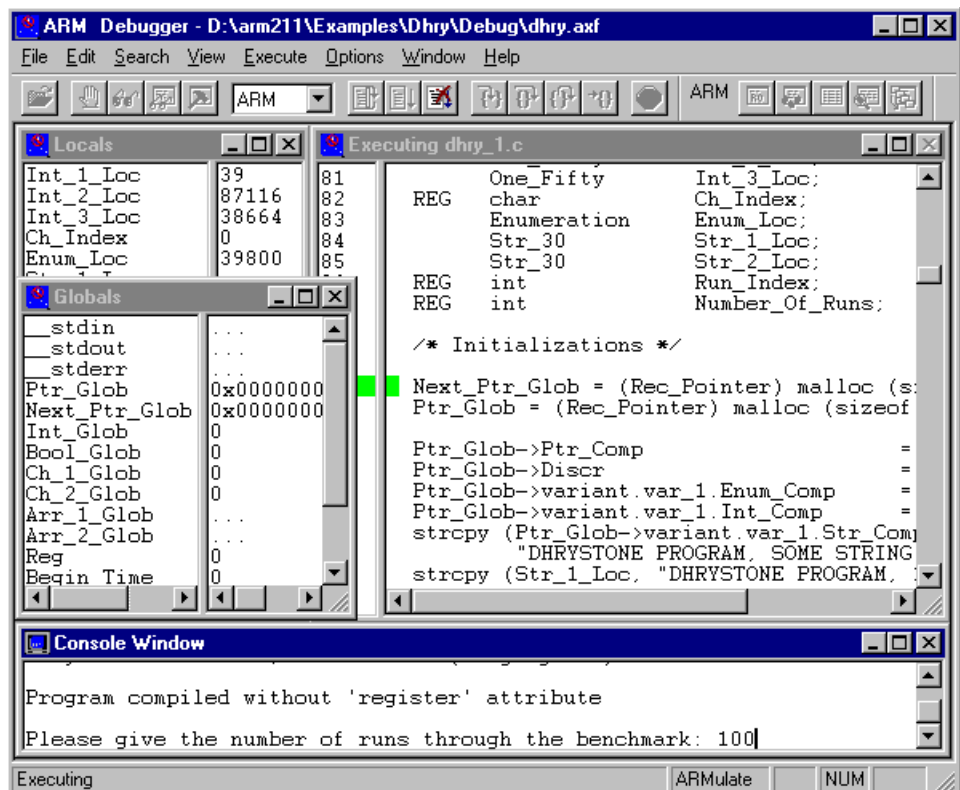
- a complex watchpoint that:
  - stops when the variable has changed a specified number of timesAND/OR
  - stops when the variable is set to a specified value

**Note** *If you set a watchpoint on a local variable, you will lose the watchpoint as soon as you leave the function which uses the local variable.*

## 3.5 The ADW Desktop

This section describes what ADW looks like and the windows that are available during your debugging session.

The ADW Desktop consists of a number of windows that are used to display a variety of information as you work through the process of debugging your executable image. Three windows, the Execution Window, the Command Window and the Console Window are opened automatically when you start the debugger and remain. The Execution Window is always open. You can open other windows by selecting the appropriate item from the **View** menu. The figure below shows ADW with the Execution, Console, Globals and Locals Windows, in the process of debugging the sample image DHRV.



# ARM Debugger for Windows

## 3.5.1 Menu bar, toolbar, mini toolbar and status bar

The menu bar at the top of the ADW Desktop; click on an item to display the pull down menu. Underneath the menus is the toolbar; position the cursor over a menu icon and a brief description will be displayed. There is also a processor specific mini toolbar. The menus, the toolbar and the mini toolbar are described in greater detail in the on-line help.

At the bottom of the Desktop is the status bar, which provides current status information or describes the currently selected user interface component.

## 3.5.2 Window-specific menus

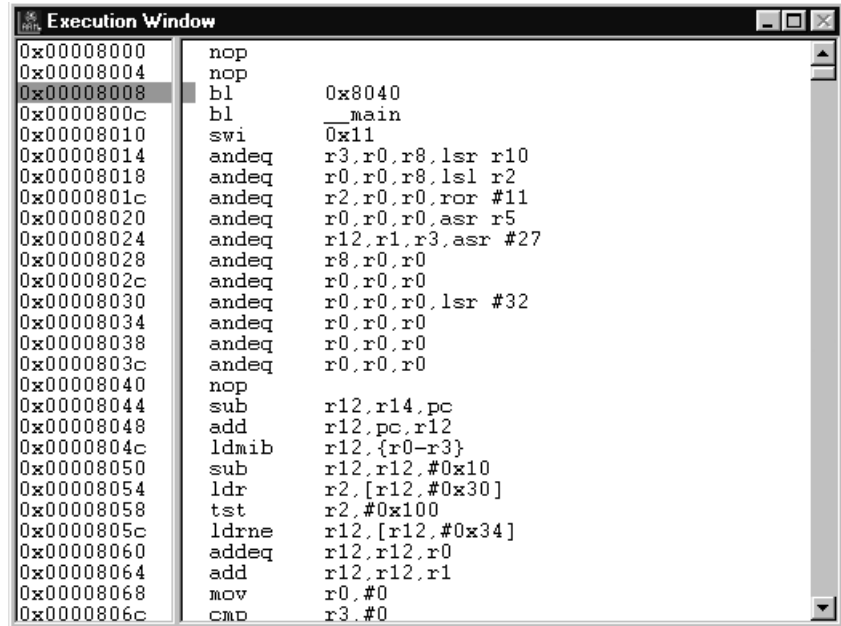
Each of ADW's windows has its own menu that is displayed when you click the secondary mouse button (typically the right mouse button), over the window. Item specific options require that you position the cursor over an item in the window before they are activated.

Each of the window-specific menus is described in the on-line help for that window menu.

## 3.5.3 The main ADW windows

The main ADW windows are the Execution Window, the Console Window and the Command Window. These three windows are always opened when you start ADW.

### Execution Window



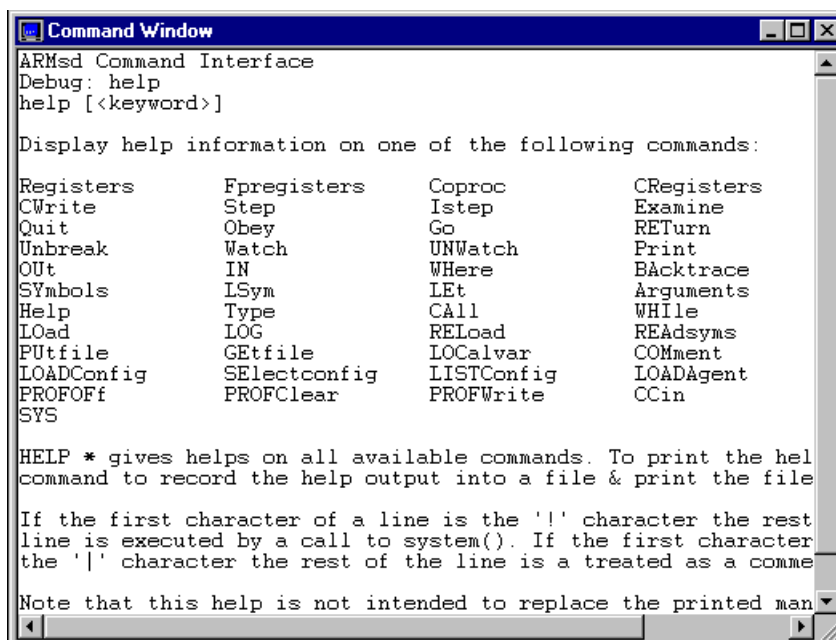
```
0x00008000    nop
0x00008004    nop
0x00008008    bl      0x8040
0x0000800c    bl      __main
0x00008010    swi     0x11
0x00008014    andeq   r3,r0,r8,lsr r10
0x00008018    andeq   r0,r0,r8,lsr r2
0x0000801c    andeq   r2,r0,r0,ror #11
0x00008020    andeq   r0,r0,r0,asr r5
0x00008024    andeq   r12,r1,r3,asr #27
0x00008028    andeq   r8,r0,r0
0x0000802c    andeq   r0,r0,r0
0x00008030    andeq   r0,r0,r0,lsr #32
0x00008034    andeq   r0,r0,r0
0x00008038    andeq   r0,r0,r0
0x0000803c    andeq   r0,r0,r0
0x00008040    nop
0x00008044    sub     r12,r14,pc
0x00008048    add     r12,pc,r12
0x0000804c    ldmib   r12,{r0-r3}
0x00008050    sub     r12,r12,#0x10
0x00008054    ldr     r2,[r12,#0x30]
0x00008058    tst     r2,#0x100
0x0000805c    ldrne   r12,[r12,#0x34]
0x00008060    addeq   r12,r12,r0
0x00008064    add     r12,r12,r1
0x00008068    mov     r0,#0
0x0000806c    cnd     r3,#0
```

# ARM Debugger for Windows

The Execution Window displays the source code of the program that is currently executing. In the Execution Window you can:

- execute the entire program or step through the program line by line
- examine the contents of variables or registers
- change the display mode to show disassembled machine code interleaved with high-level C source code
- display another area of the code by address
- set, edit or remove breakpoints

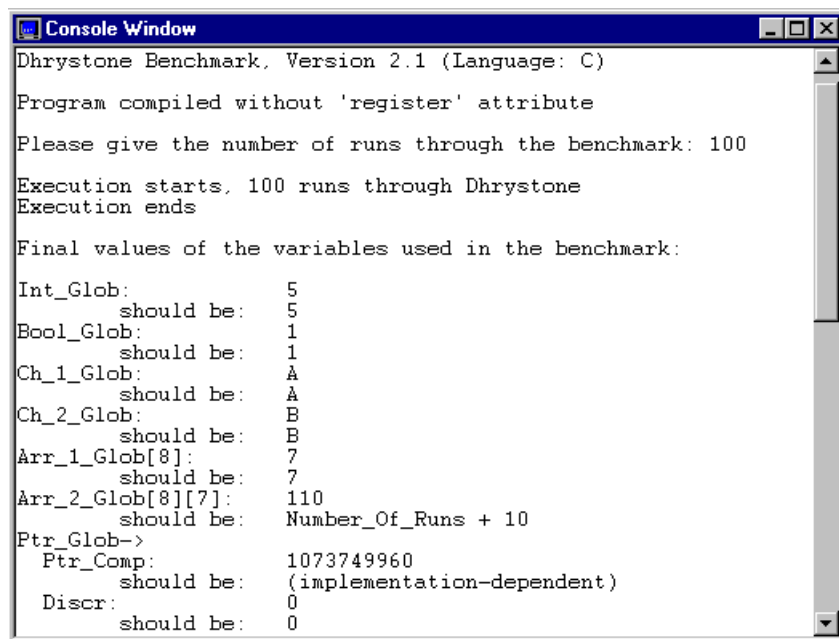
## Command Window



Using the Command Window, you can use armsd instructions when you are debugging your image. Type `help` at the Debug prompt for information on the available commands or refer to the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

# ARM Debugger for Windows

## Console Window



```
Console Window
Dhrystone Benchmark, Version 2.1 (Language: C)

Program compiled without 'register' attribute

Please give the number of runs through the benchmark: 100

Execution starts, 100 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:          5
    should be:    5
Bool_Glob:         1
    should be:    1
Ch_1_Glob:         A
    should be:    A
Ch_2_Glob:         B
    should be:    B
Arr_1_Glob[8]:     7
    should be:    7
Arr_2_Glob[8][7]: 110
    should be:    Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:        1073749960
    should be:    (implementation-dependent)
  Discr:           0
    should be:    0
```

The Console Window allows interaction between you and the executing program. Anything printed by the program (for example a prompt for user input) is displayed in this window and any input required by the program must be entered here. Information remains in the window until you select **Clear** from the **Console Window** menu. You can also save the contents of the Console Window to disk, by selecting **Save** from the **Console Window** menu.

Initially the Console Window displays the start-up messages of your target processor, eg. the ARMulator, ARM Development Board or EmbeddedICE.

**Note** *When input is required by your executable image, most ADW functions are disabled until the required information has been entered.*

## 3.5.4 Other available windows

Additional windows can be displayed from the **View** menu.

### Backtrace Window

The Backtrace Window displays current backtrace information about your program. From the Backtrace Window you can:

- show disassembly code for the current procedure
- show a list of local variables for the current procedure
- set or remove breakpoints

### Breakpoints Window

The Breakpoints Window displays a list of all breakpoints set in your image. The actual breakpoint is displayed in the right-hand pane. If the breakpoint is on a line of code, the relevant source file is shown in the left-hand pane.

From this window you can:

- show source/disassembly code
- set, edit or remove breakpoints

### Debugger Internals Window

This window displays some of ADW's internal variables. Use this window to change the value of editable variables while.

<code>\$clock</code>	Number of microseconds since simulation started.
<code>\$cmdline</code>	Argument string for the image being debugged.
<code>\$echo</code>	Non zero if commands from obeyed files should be echoed (initially set to 0).
<code>\$examine_lines</code>	Default number of lines for examine command (initially set to 8).
<code>\$format</code>	Default format for printing integer values (initially set to %d).
<code>\$fpreult</code>	Floating-point value returned by last called function (junk if none, or if a floating point value was not returned). This variable is read-only.
<code>\$inputbase</code>	Base for input of integer constants (initially set to 10).
<code>\$list_lines</code>	Default number of lines for list command (initially set to 10).

# ARM Debugger for Windows

\$pr\_linelength

\$rdi\_log

Default number of characters per line (initially set to 72).  
RDI logging:

Bit 1	Bit 0	
0	0	Off
0	1	RDI on
1	1	Device Driver Logging on
1	1	RDI and Device Logging on

Table 3-1: RDI Logging

The remaining bits should be set to zero.

\$result

\$sourcedir

\$statistics

\$statistics\_inc

\$statistics\_inc\_w

\$top\_of\_memory

\$type\_lines

\$vector\_catch

Integer result returned by last called function (junk if none, or if an integer result was not returned). This variable is read-only.  
Directory containing source code for the program being debugged. Initially set to the current directory unless your application was built by the ARM Project Manager, in which case this variable points to the source directory known by APM (initially set to NULL).  
This variable can be used to output any statistics which the ARMulator has been keeping. This variable is read-only.  
Not applicable to the Windows environment.  
This variable is similar to \$statistics, but outputs the difference between the current statistics and the point at which you asked for the \$statistics\_inc\_w window. This variable is read only.  
Under Angel, this variable gives the total amount of memory normally on the board. If more memory is added to the board, change this variable to reflect the new amount of memory.  
Default number of lines for the type command.  
Indicates whether or not execution should be caught when various conditions arise. The default value is %RUsPDaIfE. Capital letters indicate that the condition is to be intercepted:  
R                   reset





U	undefined instruction
S	SWI
P	prefetch abort
D	data abort
A	address
I	normal interrupt
F	fast interrupt
E	unused

You can also set the variable to a numeric value which will be interpreted as a bitmap, in the order set above.

## Disassembly Window

The Disassembly Window displays disassembled code interpreted from a specified area of memory. The memory addresses are listed in the left-hand pane and the disassembly code is displayed in the right-hand pane. ARM code, Thumb code or both can be displayed.

From this window you can:

- go to another area of memory.
- change the disassembly mode to ARM, Thumb or Mixed.
- set, edit or remove breakpoints.

**Tip** *More than one Disassembly Window can be active at a time.*

For more information on displaying disassembled code, see **3.8.3 Disassembly code** on page 3-37.

## Expression Window

The Expression Window is used to display the values of selected variables and/or registers.

In the Expression Window you can:

- change the format of selected items or all items.
- display information on a selected item's type.
- display the section of memory pointed to by the contents of a variable.

For more information on displaying variable information, see **3.8.2 Variables** on page 3-35.

## Function Names Window

The Function Names Window displays a list of the functions that are part of your program.

In the Function Names Window you can:

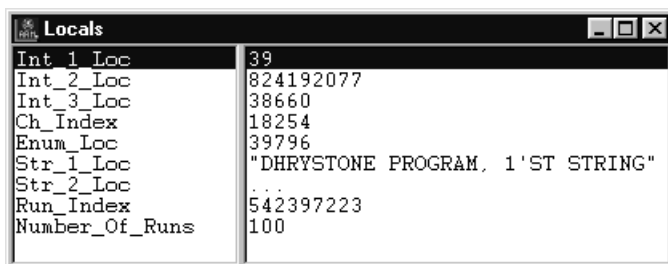
- display a selected function as source code.
- set, edit or remove a breakpoint on a function.

# ARM Debugger for Windows

---

## Locals/Globals Windows

The Locals Window displays a list of variables currently in scope and the Globals Window displays a list of global variables. The variable name is displayed in the left-hand pane, the value is displayed in the right-hand pane.



In the Locals/Globals Windows you can:

- display the section of memory pointed to by the variable.
- change the format of the values displayed by line or for the entire window. (Once the format of a line is changed, it won't be effected by changing the format of the window.)
- set, edit or remove a watchpoint on a variable.
- double-click on an item to expand a structure; the details will be displayed in another variable window.
- double-click on a value (right-hand pane) to modify a variable.

As you step through the program the variable values are updated.

For more information on displaying variable information, see **3.8.2 Variables** on page 3-35.

## Low Level Symbols Window

The Low-level Symbols Window displays a list of all the low-level symbols in your program.

In the Low-level Symbols Window you can:

- display the memory pointed to by the selected symbol.
- display the source/disassembled code pointed to by the selected symbol.
- set, edit or remove a breakpoint on the line of code pointed to by the selected symbol.

## Memory Window

This window displays the contents of memory at a specified address. Addresses are listed in the left-hand pane, and the memory content is displayed in the right-hand pane.

In the Memory Window you can:

- display other areas of memory by scrolling or specifying an address.
- set, edit or remove a watchpoint.

## Registers Window

This window displays the registers corresponding to the mode named at the top of the window, with the contents displayed in the right-hand pane. You can double-click on an item to modify the register's value.

In this window you can:

- display the contents of the register memory.
- edit the contents of a register.
- set, edit or remove a watchpoint on a register.

Multiple Register Windows can be open at any one time.

## RDI Log Window

The RDI Log Window displays remote debug information, ie. the low-level communication messages between the ARM Debugger and the target processor.

**Note** *This facility is not normally enabled, but must be specially turned on by ARM when the RDI is compiled. In addition, the Debugger internal variable `$rdi_log` must be non-zero. For more information see the ARM Software Development Toolkit Reference Guide (ARM DUI 0041).*

For more information on RDI, see **3.8.4 Remote Debug Information** on page 3-38.

## Search Paths Window

This window displays the search paths of the image currently being debugged. You can remove a search path from this window using the delete key.

# ARM Debugger for Windows

---

## Source Files List Window

The Source Files List Window displays a list of all source files that are part of the loaded image. From this window you can select a source file that will be displayed in its own Source File Window.

## Source File Window

Displays the contents of the source file named at the top of the window. The line number is displayed in the left-hand pane, the code is in the right-hand pane.

In the Source File Window you can:

- display the value of a variable, if the variable is in context.
- search for a line of code by line number.
- set, edit or remove breakpoints on a line of code.

For more information on displaying source files, see **3.8.1 Source files** on page 3-34.

## Watchpoints Window

The Watchpoints Window displays a list of all watchpoints. From this window you can:

- delete a watchpoint.
- edit a watchpoint.

## 3.6 Getting Started

This section goes through the basics of debugging an executable image, including the following debugging tasks:

- Starting the Debugger
- Loading an Image
- Executing an Image
- Stepping Through an Image
- Setting Breakpoints and Watchpoints
- Removing a Breakpoint or a Watchpoint
- Examining and Setting Variables and Registers
- Examining Memory
- Reloading the Image
- Exiting the Debugger

For information on more advanced features, see the following:

- **3.7 Debugger Configuration** on page 3-28
- **3.8 Displaying Image Information** on page 3-34
- **3.9 Setting and Editing Complex Breakpoints and Watchpoints** on page 3-39
- **3.10 Other Debugging Functions** on page 3-42
- the ADW on-line help

### 3.6.1 Starting ADW



You can start the ADW in three ways:

- 1 If you are running Windows 95 or NT4, select ARM Debugger for Windows from the ARM SDT V2.11 Program folder on the Start menu.
- 2 If you are running Windows NT3.51, double-click on the ARM Debugger icon in the ARM SDT V2.11 Program group.
- 3 If you are working in the ARM Program Manager, click the ARM Debugger button or select **Debug project** from the **Project** menu.



The Console, Command, and Execution Windows are displayed, and you can go on to load your executable image.

# ARM Debugger for Windows

---

## 3.6.2 Loading an image



When you load a program image, the program is displayed in the Execution Window as disassembly code (the Command and Console Windows are also displayed), and a breakpoint is automatically set at the entry point of the image, usually the first line of source after the `main()` function. The current execution marker (a green bar indicating the current line) is located at the entry point of the program.

**Note** *Once you have executed your program you must reload it to execute again, using the **Reload** button or by selecting **Reload** from the **File** menu (see 3.6.9 Reloading the image on page 3-27)*

- 1 Choose **Load Image** from the **File** menu or click the **Open File** button. The Open File dialog is displayed.
- 2 Select the **File Name** of the executable image you wish to debug, using the **Browse** option if necessary.
- 3 Enter any command line **Arguments** expected by your image.
- 4 Click **OK**.

Alternatively, if you have recently loaded your required image, your file appears as a recently used file on the File menu.

See also **3.10.5 Flash Download** on page 3-45.

**Note** *If you load your image from the recently used file list, the ARM Debugger automatically loads the image using the command-line arguments you specified in the previous run.*

## 3.6.3 Executing an image



You can run your program in ADW in one of two ways. First, you can execute the entire program; ADW will halt execution at any breakpoints or watchpoints encountered. Second, you can step through the code a line at a time, stepping into or over procedures as needed. Stepping through your image is covered in the next section. To execute your image:

- Select **Go** from the **Execute** menu.

OR

- Click the **Go** button.

While the program executes, the Console Window is activated and the program code is displayed in the Execution Window.

Execution continues until:

- a breakpoint halts the program at a specified point.
- a watchpoint halts the program when a specified variable or register changes.
- the image requires input.
- you stop the program by clicking the **Stop** button. You can then continue execution from the point where the program stopped using **Go** or **Step**.



If the program is failing to respond, you can also abort program execution.

- Note** *If you wish to execute your program again, you must reload it first.*
- Tip** *If you are in the ARM Project Manager, you can click the **Execute** button; the image will be built if necessary, the Debugger will be started, and your image will be loaded and executed (see **Executing an image** on page 2-18).*

## 3.6.4 Stepping through an image

If you want to more closely follow your image's execution, you can step through the code in the following ways:



### Step to the next line of code:

- Select **Step** from the **Execute** menu.

OR

- Click the **Step** button.

The program moves to the next line of code, which is highlighted in the Execution Window. Function calls will be treated as one statement.

If only C code is displayed, **Step** moves to the next line of C. If disassembled code is interleaved with C source, **Step** moves to the next line of disassembled code.



### Step In to a function call:

- Select **Step In** from the **Execute** menu.

OR

- Click the **Step In** button.

The program moves to the next line of code. If the code is in a called function, the function source is displayed in the Execution Window, and the current code line is highlighted.



### Step Out of a function

- Select **Step Out** from the **Execute** menu.

OR

- Click the **Step Out** button.

The program completes execution of the function and halts at the line immediately following the function call.



### Run execution to the cursor:

- 1 Position the cursor in the line where execution should stop.
- 2 Select **Run to Cursor** from the **Execute** menu or click the **Run to Cursor** button.

This executes the code between the current execution and the position of the cursor.

- Note** *Be sure that the execution path includes the statement selected with the cursor.*

# ARM Debugger for Windows

---

## 3.6.5 Setting breakpoints and watchpoints

Breakpoints and watchpoints are used to stop program execution when a selected line of code is about to be executed or when a specified condition occurs. There are two types of breakpoints and watchpoint; simple and complex. This section discusses simple breakpoints and watchpoints; complex breakpoints and watchpoints are discussed in **3.9 Setting and Editing Complex Breakpoints and Watchpoints** on page 3-39.



### Breakpoints

To set a simple breakpoint on a line of code:

- Double-click on the line where you want to set the breakpoint.

OR

- 1 Position the cursor in the line where the breakpoint is to be placed.
- 2 Select **Toggle Breakpoint** from the **Execute** menu or click the **Toggle breakpoint** button.

A new breakpoint is displayed as a red marker in the left-hand pane of the Execution Window, the Disassembly Window or the Source File Window. If the line in which the breakpoint is set contains several functions, the breakpoint is set on the function which you clicked on in step 1.

In a line with several statements, it is possible to set a breakpoint on an individual statement, as demonstrated in the following example:

```
int main()  
{  
    hello(); world();  
}
```

If you position the cursor on the word “world” and click the **Toggle breakpoint** button, `hello` will be executed, but execution will halt before `world` is executed.

If you want to see all of the breakpoints set in your executable image, open the Breakpoints Window by selecting **Breakpoints** from the **View** menu.

To set a simple breakpoint on a function:

- 1 Display a list of function names in the Function Names Window, by selecting **Function Names** from the **View** menu.
- 2 Select **Toggle Breakpoint** from the **Function Names Window** menu or click the **Toggle breakpoint** button.

The breakpoint will be set at the first statement of the function. This method also works for the Low Level Symbols Window, but the breakpoint will be set at the first machine instruction of the function (ie. the beginning of its entry sequence).





## Watchpoints

To set a simple watchpoint:

- 1 Select the variable, area of memory or register you want to watch.
- 2 Select **Toggle Watchpoint** from the **Execute** menu.

OR

- Select the **Toggle Watchpoint** option from the window's menu.

OR

- Click the **Watchpoint** button.

If you want to see all of the watchpoints set in your executable image, open the Watchpoints Window by selecting **Watchpoints** from the **View** menu.

### 3.6.6 Removing a breakpoint or watchpoint

To remove a breakpoint:

- 1 Double-click on a line containing a breakpoint (highlighted in red) in the Execution Window.
- 2 Select **Toggle Watchpoint** from the menu.

OR

- 1 Select **Breakpoints** from the **View** menu to display a list of breakpoints in the Breakpoint Window.
- 2 Select the breakpoint you wish to remove.
- 3 Click the **Toggle breakpoint** button or press the **Delete** key.

To remove a watchpoint:

- 1 Select **Watchpoints** from the **View** menu to display a list of watchpoints in the Watchpoint Window.
- 2 Select the watchpoint you wish to remove.
- 3 Click the **Toggle watchpoint** button or press the **Delete** key.

OR

- 1 Position the cursor on a variable or register containing a watchpoint and right click.
- 2 Select **Toggle Watchpoint** from the menu.

**Note** *If you set a watchpoint on a local variable, you will lose the watchpoint as soon as you leave the function which uses the local variable.*

# ARM Debugger for Windows

---

## 3.6.7 Examining and setting variables and registers

Using the Debugger, you can display and modify the contents of the variables and registers used by your executable image. This section briefly introduces the display and modification features, see **3.8.2 Variables** on page 3-35, for more information on variables and registers.

### Variables

To display a list of variables



- Select **Local** from the **Variables** sub-menu off the **View** menu or click the **Locals** button.

OR

- Select **Global** from the **Variables** sub-menu off the **View** menu.

A Locals or Globals Window is displayed listing the variables currently active.

To modify a variable's value:

- 1 Select **Global** or **Local** from the **Variables** sub-menu off the **View** Menu. A Locals/ Globals Window is displayed with the currently active variables.
- 2 Double-click on the value of the variable in the right-hand pane of the window. The Modify Item dialog is displayed.
- 3 Type in the new value for the variable.
- 4 Click **OK**.

### Registers

To display a list of registers:

- Select a mode from the **Registers** sub-menu off the **View** menu. The registers are displayed in the appropriate Registers Window.
- To display a list of registers for User mode, click the **User Regs** button.



To modify a register's value:

- 1 Select a register mode from the **Register** sub-menu off the **View** menu. The registers for that mode are displayed in a Registers Window.
- 2 Double-click on the register to be modified. The Modify Item dialog is displayed.
- 3 Type in the new value for the register.
- 4 Click **OK**.

## 3.6.8 Examining memory



Using the Debugger, you can display memory locations.

To display a particular area of memory:

- 1 Select **Memory** from the **View** menu or click on the **Memory** button. The Memory Address dialog is displayed.
- 2 Enter the address as hex (prefixed by 0x) or decimal.
- 3 Click **OK**.

The Memory Window is opened to display the area of memory requested.

Once you have opened the Memory Window you can display other areas of memory by using the scroll bars or by entering another address.

To enter another address:

- 1 Select **Goto** from the **Search** menu or select **Goto Address** from the **Memory Window** menu. The Goto Address dialog is displayed
- 2 Enter an address.
- 3 Click **OK**.

See **3.10.1 Saving or changing an area of memory** on page 3-42 for more information on working with areas of memory.

## 3.6.9 Reloading the image



Once you have executed your image, if you want to execute it again, you must reload it. To reload your executable image, select **Reload Current image** from the **File** menu or click the **Reload** button.

## 3.6.10 Exiting the Debugger

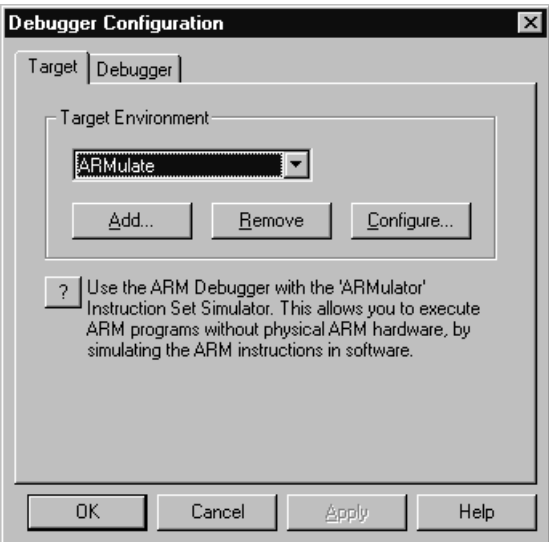
To close the Debugger, select **Exit** from the **File** menu.


# ARM Debugger for Windows

## 3.7 Debugger Configuration

### 3.7.1 Target

Use this dialog to change the configuration used by the target environments that will be used during debugging. Accessed by selecting **Configure Debugger** from the **Options** menu.



<b>Target Environment</b>	The target environment for the image being debugged.
<b>Add</b>	Display an Open dialog to add a new environment to the debugger configuration.
<b>Remove</b>	Remove a target environment.
<b>Configure</b>	Display a configuration dialog for the selected environment.
	Display a more detailed description of the selected environment.

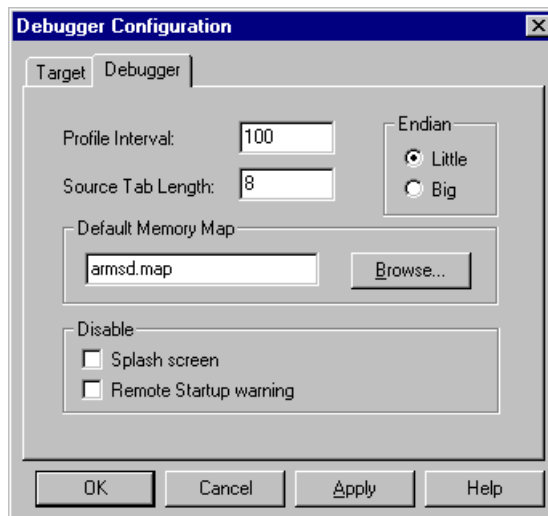
When your changes are complete:

- click **OK** to save and exit.
- click **Cancel** to ignore all changes not applied and exit.

**Note** **Apply** is disabled for the **Target** page because a successful RDI connection has to be made first. When you click **OK** an attempt is made to make your selected RDI connection, if this isn't successful the **ARMulate** setting will be restored.

## 3.7.2 Debugger

Use this dialog to change the configuration used by the Debugger. Accessed by selecting **Configure Debugger** from the **Options** menu and clicking the **Debugger** tab.



- |                              |   |
|------------------------------|---|
| <b>Profile Interval</b>      | Time between PC-sampling in microseconds. Lower values have a higher performance overhead, and will slow down execution, but higher values are not as accurate. |
| <b>Source Tab Length</b>     | When a source file is displayed this specifies the length of tab used in characters.  |
| <b>Default Memory Map</b>    | The default memory map, the file which describes your memory layout (see <b>8.3.5 Map files</b> on page 8-8).   |
| <b>Endian</b>                | Determines byte sex.<br>Little-endian    low addresses have the least significant bytes<br>Big-endian     high addresses have the least significant bytes       |
| <b>Disable Splash screen</b> | When check-marked, stops display of the splash screen (the ADW startup box) when ADW is first loaded.   |

# ARM Debugger for Windows

## Remote Startup warning

Turns on or off the warning that debugging is starting with Remote\_A or Remote\_D enabled. If the warning is turned off and debugging is started without the necessary hardware attached, there is a possibility that ADW may hang. If the warning is enabled, you have the opportunity to start in ARMulate.

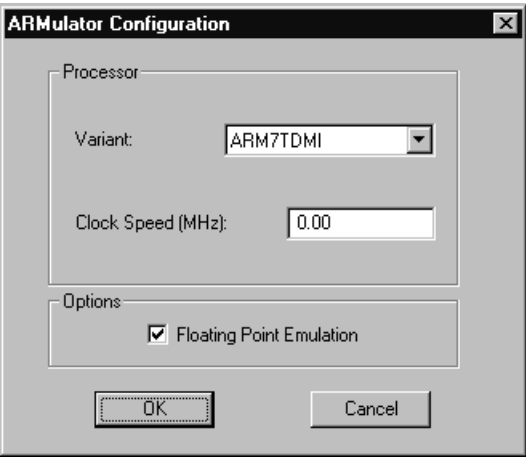
When your changes are complete:

- click **OK** to save and exit.
- click **Apply** to save.
- click **Cancel** to ignore all changes not applied and exit.

**Note** *When you make changes to the Debugger configuration the current execution is ended and your program is reloaded.*

### 3.7.3 ARMulator

Use this dialog to change configuration settings for the ARMulator. Accessed by selecting ARMulate in the **Target Environment** field on the **Target** tab of the Debugger Configuration dialog (accessed by selecting **Configure Debugger** from the **Options** menu).



**Variant**

Processor type required for emulation

**Clock speed**

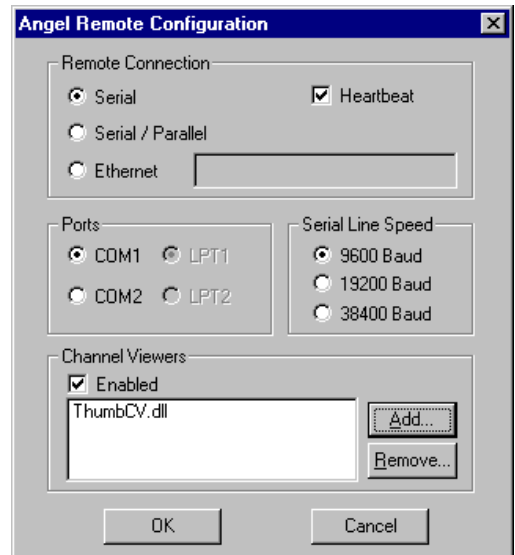
Clock speed required for emulation. If this field is set to 0.00, the real-time clock of the host computer will be used, rather than an emulated clock.

## Floating point emulation

Toggles floating point emulation on/off emulating the floating-point unit of a processor.

### 3.7.4 Remote\_A (Angel)

Use this dialog to configure the settings for the Remote\_A connection you are using to debug your application. Accessed by selecting Remote\_A in the **Target Environment** field on the Target tab of the Debugger Configuration dialog (accessed by selecting **Configure Debugger** from the **Options** menu).

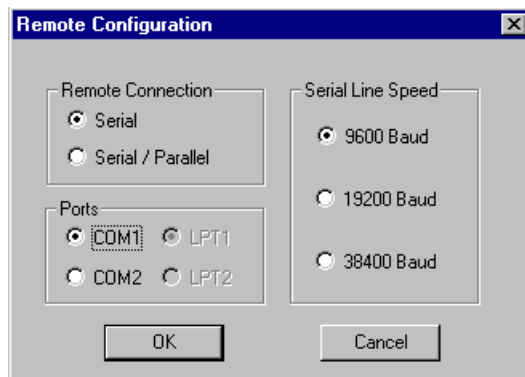


- |                          |  |
|--------------------------|--|
| <b>Remote Connection</b> | Choose either Serial or Serial/Parallel depending on the connections. For Ethernet, enter either an IP address or the hostname of the target board.  |
| <b>Heartbeat</b>         | Ensures reliable transmission by sending heartbeat messages. If not enabled, there is a danger that both host and target can get into a deadlock situation with both waiting for a packet.   |
| <b>Ports</b>             | Choose either COM1 or COM2 for you serial connection and LPT1 or LPT2 for your parallel connection.  |
| <b>Serial Line Speed</b> | Baud rate used to transmit data along the serial line.   |
| <b>Channel Viewers</b>   | Enable or disable the selected Channel Viewer DLL. See <b>3.10.6 Channel Viewers</b> on page 3-45 and <i>Application Note 38</i> (ARM DAI 0038) for more information on Channel Viewers.<br>Add        adds a Channel view DLL.<br>Remove    removes the selected DLL. |

# ARM Debugger for Windows

## 3.7.5 Remote\_D (Demon)

Use this dialog to configure the settings for the Remote\_D connection you are using to debug your application. Accessed by selecting Remote\_D in the **Target Environment** field on the **Target** tab of the Debugger Configuration dialog (accessed by selecting **Configure Debugger** from the **Options** menu).



**Remote connection** Select either Serial or Serial/Parallel depending on the connections you have to the board.

**Ports** Select either COM1 or COM2 for you serial connection and LPT1 or LPT2 for your parallel connection.

**Serial line speed** Baud rate used to transmit data along the serial line.

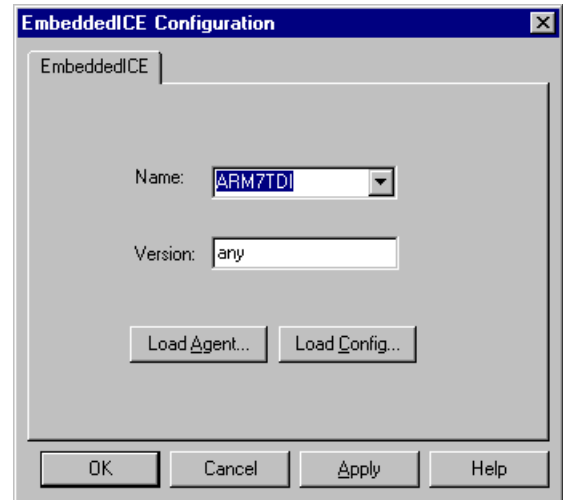
**Note** *Remote\_D has been superseded by the Angel Debug monitor (Remote\_A) – refer to Application Note 39: Demon and RDP (ARM DAI-0039).*

**Note** *Remote\_D is not shown on the target connections list on the Debugger Configuration - Target property page (page 3-28). To use Remote\_D you must add it to the list.*



## 3.7.6 EmbeddedICE Configuration

Use this dialog to select the settings for EmbeddedICE target. This option is enabled only if you have EmbeddedICE connected to your machine (accessed by selecting **Configure EmbeddedICE** from the **Options** menu).



<b>Name</b>	Name given to the EmbeddedICE configuration. Currently restricted to ARM7DI and ARM7TDI.
<b>Version</b>	Version given to the EmbeddedICE configuration. Specify the specific version to use or enter <b>any</b> if you do not require a specific implementation.
<b>Load Agent</b>	Specify a file (new EmbeddedICE ROM image) and run it; used for major updates to the ROM.
<b>Load Config</b>	Prompts for an EmbeddedICE configuration file, which is loaded. Click <b>OK</b> to run. (Generally used for minor updates.)

# ARM Debugger for Windows

---

## 3.8 Displaying Image Information

Certain information can be displayed by selecting the appropriate item from the View menu:

- Breakpoints
- Watchpoints
- Backtrace
- Functions
- Debugger Internals
- Registers

The windows used to display this information are described in **3.5 The ADW Desktop** on page 3-11.

Information not available directly from the **View** menu is discussed in this section.

### 3.8.1 Source files

#### Search paths

If you want to view the source for your program's image during the debugging session, the ARM Debugger needs to know how to find the files. A search path points to a directory or set of directories that are used to locate files whose location is not referenced absolutely.

If you are developing your program using the ARM Project Manager, the search path for a newly loaded image is added to the list of paths by reading the build directory from the image file.

If you are using the ARM command line tools to build your project, you may need to edit the search paths for your image manually, depending on the options you chose when you built it.

If for some reason the files have moved since the image was built, the search paths for these files must be set up in the ARM Debugger, using the Add Path dialog (see below).

To display source file search paths, select **Search Paths** from the **View** menu.

The current search paths are displayed in the Search Paths Window.

To add a source file search path:

- 1 Select **Add a Search Path** from the **Options** menu  
The Select a file from the required directory dialog is displayed.
- 2 **Browse** for the directory you wish to add and highlight any file in that directory.
- 3 Click **OK**.

To delete a search path:

- 1 Select **Search Paths** from the **View** menu. The Search Paths Window is displayed.
- 2 Select the path to delete.
- 3 Press the **Delete** key.

## Listing source files

To display a list of the current program's source files, select **Source Files** from the **View** menu.

The Source Files List Window is displayed.

## Source files

Once you have a listing of source files in the Source Files List Window, you can select a source file to be displayed by double-clicking on a file name.

The file is opened in its own Source File Window.

**Note** *You can have more than one source file open at a time.*

## 3.8.2 Variables

A list of local or global variables can be displayed by selecting the appropriate item from the **View** menu; a Locals/Globals Window is displayed. You can also display the value of a single variable or you can display additional variable information from the Locals/Globals Window.

To display the value of a single variable:

- 1 Select **Expression** from the **Variables** sub-menu off the **View** menu.
- 2 Enter the name of the variable in the View Expression dialog.
- 3 Click **OK**.

Alternatively:

- 1 Highlight the name of the variable.
- 2 Select **Immediate Evaluation** from **Variables** sub-menu off the **View** menu or click the **Evaluate Expression** button.



In both cases, the value of the variable is displayed in an Expression Value information box and is recorded in the Command Window.

**Note** *If you select a local variable that is not in the current context, an error message is displayed.*

# ARM Debugger for Windows

## Display formats

If you are in the Locals or the Globals Window, Expressions Window or the Debugger Internals Window, you can change the format of a variable. The format of values displayed for variables can be modified using the same syntax as a `printf` format string in C. Format descriptors include:

Type	Format	Description
int		Only use this if the expression being printed yields an integer :
	%d	Signed decimal integer (default for integers)
	%u	Unsigned integer
char	%x	Hexadecimal (lowercase letters)
		Only use this if the expression being printed yields an integer:
	%c	Character
char*	%s	Pointer to character. Only use this for expressions which yield a pointer to a zero terminated string.
void*	%p	Pointer (same as % . 8x), eg. 00018abc . This is safe with any kind of pointer.
float		Only use this for floating-point results :
	%e	Exponent notation, eg. 9 . 999999e+00
	%f	Fixed point notation, eg. 9 . 999999
	%g	General floating-point notation, eg. 1 . 1 , 1 . 2e+06

Table 3-2: Display Formats

To change the format of a variable:

- 1 Right click on the variable and select the **Change line format** from the Locals or Globals Window menu. The Display Format dialog is displayed.
- 2 Enter the display format.
- 3 Click **OK**.

**Note** If you change a single line, that line will not be effected by global changes.

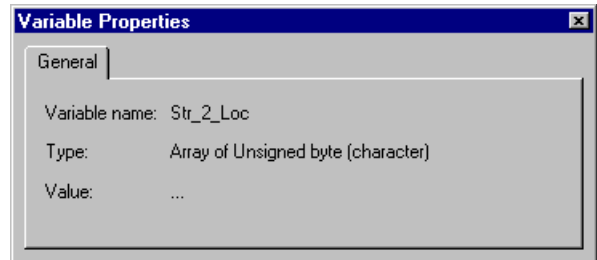
**Tip:** Leaving the Display Format dialog empty and clicking **OK** restores the default display format. This is the method to revert a line format change to the global format.



**Note** *The initial display format of a variable declared as `char[]` is special; the whole string is displayed, whereas normally arrays are displayed as ellipsis. If the format is changed it will revert to the standard array representation.*

## Variable properties

If you have a list of variables displayed in a Locals/Globals Window, you can display additional information on a variable by selecting **Properties** from the window's menu (right click on an item to display the window menu). The information is displayed in a dialog.



## Indirection

By selecting **Indirect through item** from the **Variables** menu you can display other areas of memory.

If you select a variable of integer type, the value is converted to a pointer (using sign extension where applicable) and the memory at that location is displayed. If you select a pointer variable, the memory at the location pointed to is displayed. You cannot select a void pointer for indirection.

### 3.8.3 Disassembly code

Disassembled code is a textual form of the machine code generated by the ARM C compiler or assembler.

You can display disassembled code in the Execution Window or in the Disassembly Window (select **Disassembly** from the **View** menu).

You can also choose the type of disassembled code to display by accessing the **Disassembly mode** sub-menu, which is off the **Options** menu. ARM code, Thumb code or both can be displayed, depending on your image.

To display or hide disassembled code in the Execution Window, select **Toggle Interleaving** from the **Options** menu.

Disassembled code is displayed in grey, the C code in black.

# ARM Debugger for Windows

---

To display an area of memory as disassembled code:

- 1 Select **Disassembly** from the **View** menu

OR



Click the **Display Disassembly** button. The Disassembly Address dialog is displayed.

- 2 Enter an address.
- 3 Click **OK**.

The Disassembly Window is opened to interpret the memory as disassembly code.

Once you have opened the Disassembly Window you can display another address as disassembled code by using the scroll bars to search for an address by value or:

- 1 Select **Goto** from the **Search** menu.
- 2 Enter an address.
- 3 Click **OK**.

## Specifying a disassembly mode

The ARM Debugger tries to interpret whether disassembled code is ARM code or Thumb code, but sometimes this is not possible, for example if you have copied the contents of a file on disk to memory.

To specify the type of code (ARM, Thumb or both) you want to see when you display disassembled code in the Execution Window, select **Disassembly mode** from the **Options** menu.

### 3.8.4 Remote Debug Information

The RDI Log Window displays remote debug information, ie. the low-level communication messages between the ARM Debugger and the target processor.

This facility is not normally enabled, but must be specially turned on by ARM when the RDI is compiled. For more information see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

To display remote debug information (RDI) select **RDI Protocol Log** from the **View** menu. The RDI Log Window is displayed.

Using the RDI Log Level dialog (select **Set RDI Log Level** from the **Options** menu) you can select the information that will be displayed in the RDI Log Window:

- |       |                              |
|-------|------------------------------|
| Bit 0 | RDI level logging on/off     |
| Bit 1 | Device driver logging on/off |

#### Warning

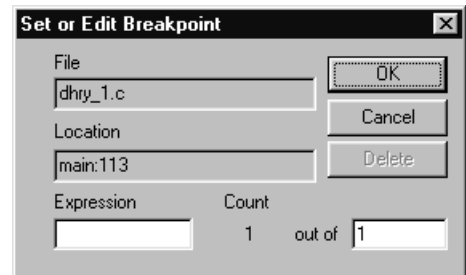
*The RDI log level is used internally within ARM to assist with debugging, this level should be changed only if you have been requested to do so by ARM.*

## 3.9 Setting and Editing Complex Breakpoints and Watchpoints

### 3.9.1 Breakpoints

A breakpoint is a point in the code where your program will be halted by the ARM Debugger. Once you have set a breakpoint it will appear as a red marker in the left-hand pane of a Source or Execution window.

When you set a complex breakpoint, you specify additional conditions in the form of expressions entered in the Set or Edit Breakpoint dialog.



This dialog contains the following fields:

<b>File</b>	The source file that contains the breakpoint. This field is read-only.
<b>Location</b>	The position of the breakpoint within the source file. This field is read-only.
<b>Expression</b>	An expression that must be true for the program to halt, in addition to any other breakpoint conditions. Use C-like operators such as: $i < 10$ $i != j$ $i != j + k$
<b>Count</b>	The program halts when all the breakpoint conditions apply for the <i>n</i> th time.

#### To set or edit a complex breakpoint on a line of code:

- 1 Double-click on the line where you want to set a breakpoint or on an existing breakpoint position.  
The Set or Edit Breakpoint dialog is displayed.
- 2 Enter or alter the details of the breakpoint.
- 3 Click **OK**.

# ARM Debugger for Windows

The breakpoint is displayed as a red marker in the left-hand pane of the Execution, Source File or Disassembly Window. If the line in which the breakpoint is set contains several functions, the breakpoint is set on the function which you highlighted in step 1.

**To set or edit a complex breakpoint on a function:**

- 1 Display a list of function names in the Function Names Window.
- 2 **Select Set or Edit Breakpoint** from the **Function Names Window** menu.
- 3 The Set or Edit Breakpoint dialog is displayed. Complete or alter the details of the breakpoint.
- 4 Click **OK**.

**To set a breakpoint on a low-level symbol:**

- Type `break@symbolname` in the Command Window.

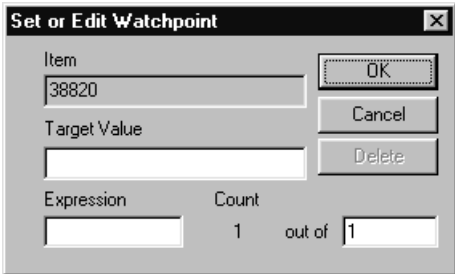
OR

- Display the Low Level Symbols Window and set a breakpoint on the required symbol.

## 3.9.2 Watchpoints

A watchpoint halts a program when a specified register or variable is changed.

When you set a complex watchpoint, you specify additional conditions in the form of expressions entered in the Set or Edit Watchpoint dialog.



This dialog contains the following fields:

- |                     |   |
|---------------------|---|
| <b>Item</b>         | The variable or register to be watched.   |
| <b>Target Value</b> | The value of the variable or register that will cause the program to halt. If this value is not specified, any change in the item's value will cause the program to halt, dependent on the other watchpoint conditions. |



<b>Expression</b>	Any expression which must be true for the program to halt, in addition to any other watchpoint conditions. As with breakpoints, use a C-like operators such as:  $i < 10$ $i \neq j$ $i \neq j + k$
<b>Count</b>	The program halts when all the watchpoint conditions apply for the <i>n</i> th time.

## To set a complex watchpoint:

- 1 Select the variable or register you want to watch.
- 2 Select **Set or Edit Watchpoint** from the **Execute** menu.  
The Set or Edit Watchpoint dialog is displayed.
- 3 Specify the details of the watchpoint.
- 4 Click **OK**.

## To edit a complex watchpoint:

- 1 Display current watchpoints by selecting **Watchpoints** from the **View** menu.
- 2 Select the watchpoint you want to edit.
- 3 Modify the details as required.

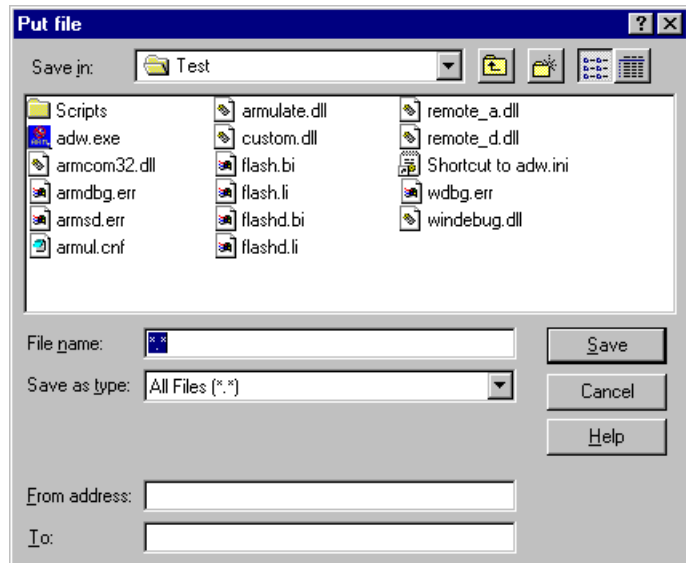
# ARM Debugger for Windows

## 3.10 Other Debugging Functions

### 3.10.1 Saving or changing an area of memory

To save an area of memory to a file on disk:

- 1 Select **Put File** from the **File** menu. The Put file dialog is displayed.

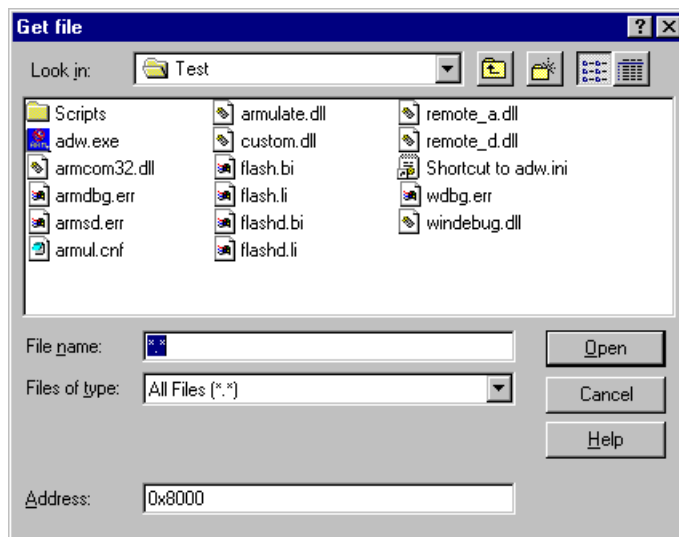


- 2 Select the file to write to.
- 3 Enter a an memory area in the **From address** and **To** fields.
- 4 Click **Save**.
- 5 Click **OK**.

**Note** The output is saved as a binary data file.

To copy a file on disk to memory:

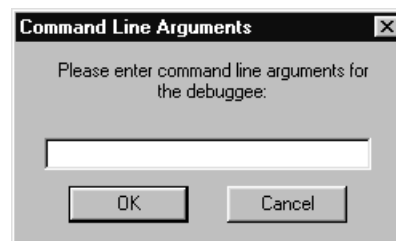
- 1 Select **Get File** from the **File** menu. The Get file dialog displayed.



- 2 Select the file you want to download.
- 3 Enter an address where the file should be loaded.
- 4 Click **Open**.

### 3.10.2 Specifying command line arguments for your program

- 1 Select **Set Command Line Args** from the **Options** menu. The Command Line Arguments dialog is displayed.



- 2 Enter the command line arguments for your program.
- 3 Click **OK**.

**Note** You can also specify command line arguments when you load your program in the Open File dialog or by changing the Debugger internal variable, \$cmdline.

# ARM Debugger for Windows

---

## 3.10.3 Using Command Line Debugger instructions

If you are used to using the ARM Command Line Debugger you may prefer to use the same set of commands from the Command Window.

To open this window select **Command** from the **View** menu.

The Command Window displays a `Debug:` command line. You can enter ARM Command Line Debugger commands at this prompt. The syntax used is the same as that used for `armsd`. Type `help` for information on the available commands.

Refer to **Chapter 4, Command-Line Development** and the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for more information on the Command Line Debugger.

## 3.10.4 Profiling

Profiling allows the programmer to see where most of the processor time is spent within a program, by sampling the PC at time intervals set by the user. This information is then used to build up a picture of the percentage time spent in each procedure. Using the command line program `armprof` on the data generated by either `armsd` or `ADW`, the programmer can see where effort can be most effectively spent to make the program more efficient.

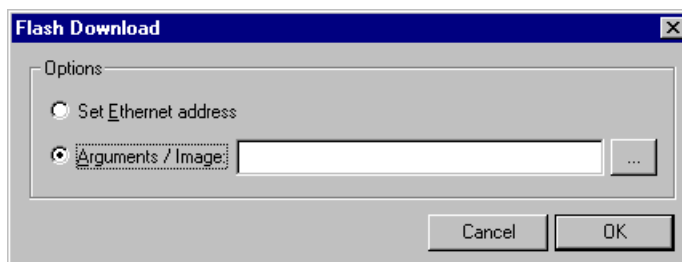
- 1 Load your image file.
- 2 Select **Toggle Profiling** from the **Profiling** sub-menu off the **Options** menu.
- 3 Execute your program.
- 4 When the image terminates, select **Write to File** from the **Profiling** sub-menu off the **Options** menu. A Save dialog appears.
- 5 Enter a file name and a directory as necessary.
- 6 Click **Save**.

**Tip** Once you have started program execution you cannot turn profile collection on. However, if you want to collect information on only a certain part of the execution, you can initiate collection before executing the program, clear the information collected up to a certain point (for example at a breakpoint), by selecting **Clear Collected** from the **Profiling** sub-menu off the **Options** menu, then execute the remainder of your program.

See **Chapter 8, Benchmarking, Performance Analysis and Profiling** for more information on profiling.

## 3.10.5 Flash Download

The Flash Download dialog is used to write an image to the flash memory chip on an ARM Development Board or any suitably equipped hardware.



**Set Ethernet Address** After writing an appropriate image to the flash memory ie. (Angel with ethernet support), this option will set the ethernet address (not necessary if you have built your own Angel with address compiled in). When you click **OK**, you will be prompted for the IP address and netmask (ie. 193.145.156.78)



**Arguments / Image** Specifies the arguments or image to write to flash. Use the **Browse** button to select the image.

**Note** You can build your own flash image using the example in the Target Development System User Guide (ARM DUI 0061). ADW will look for one of the following images, `flash.li`, `flash.bi`, `flashd.li`, `flashd.bi`, depending on whether you are using Angel, Demon, big- or little-endian. It will look first in the current working directory and then in the Toolkit bin directory.

## 3.10.6 Channel Viewers

Debug communication channels can be accessed using a Channel Viewer. An example channel viewer is supplied with ADW (`ThumbCV.dll`) or you can provide your own viewer.

To select a Channel Viewer:

- 1 Select **Configure Debugger** from the **Options** menu.
- 2 On the Target tab, select **Remote\_A**.
- 3 Click the **Configure** button. The Angel Remote Configuration dialog is displayed.
- 4 Select the **Channel Viewer Enabled** option. The **Add** and **Delete** buttons are activated.
- 5 Click the **Add** button and a list of .DLLs will be displayed.
- 6 Select the appropriate .DLL and click the **Open** button.
- 7 Clicking the **OK** button on both the Angel Remote Configuration dialog and the Debugger Configuration dialog will cause ADW to restart with an active channel viewer.

# ARM Debugger for Windows

---

See **3.3.3 Remote\_A** on page 3-4 for more information on the Remote\_A Configuration dialog box.

ThumbCV.DLL provides the following viewer:



The window has a dockable dialog bar at the bottom, this is used to send information down the channel. Typing information in the edit box and clicking the **Send** button will store the information in a buffer, the information is then sent when requested by the target. The Left to send counter displays the number of bytes that are left in the buffer.

## **Sending information**

To send information to the target, type a string into the edit box on the dialog bar and click the **Send** button. The information is then sent when requested by the target, in ASCII character codes.

## **Receiving information**

The information that is received by the channel viewer is converted into ASCII character codes and displayed in the window, if the channel viewers are active. However, if `0xffffffff` is received the following word will be treated and displayed as a number.

# 4

## Command-Line Development

4.1	Introduction	4-2
4.2	The Hello World Example	4-2
4.3	armsd	4-7

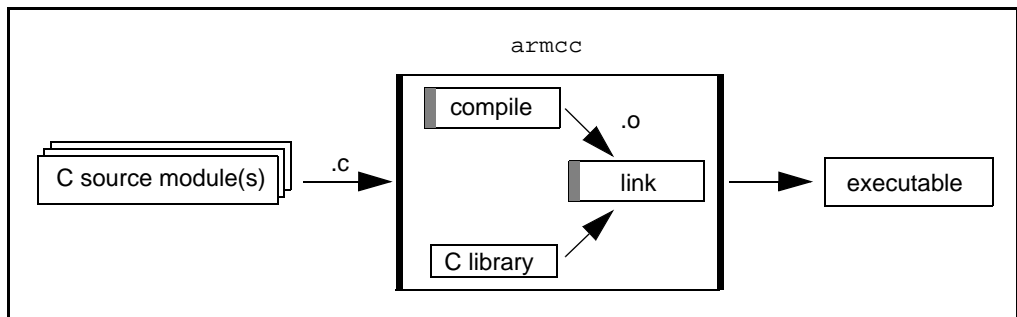
# Command-Line Development

## 4.1 Introduction

Although you can construct, build and debug your applications using the command-line tools discussed in the previous chapters, command-line tools are still available. This section gives a brief overview of using the command-line tools, for more information, please refer to the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

## 4.2 The Hello World Example

This example shows you how to write, compile, link and execute a simple C program that prints "Hello World" and a carriage return on the screen. The code will be entered using a text editor, compiled and linked using `armcc`, and run on `armsd`. This section also provides a brief introduction to `armsd`, more information is given in **4.3 `armsd`** on page 4-7.



**Figure 4-1: Compiling and linking C**

### 4.2.1 Create, compile, link, and run

Enter the following code using any text editor, and save the file as `hello.c`.

```
#include <stdio.h>

int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

Use the following command to compile and link the code:

```
armcc hello.c -o hello
```

The argument to the `-o` flag gives the name of the file that will hold the final output of the link step. The linker is automatically called after compilation (because in this instance the `-c` flag has not been specified). Note that flags are case-sensitive.



To execute the code under software emulation, enter:

```
armsd hello
```

at the system prompt. `armsd` will start, load in the file, and display the `armsd:` prompt to indicate that it is waiting for a command. Type:

```
go
```

and press **Return**. The debugger should respond with “Hello World”, followed by a message indicating that the program terminated normally.

To load and run the program again, enter:

```
reload
```

```
go
```

To quit the debugger, enter:

```
quit
```

## 4.2.2 Debugging hello.c

Next, re-compile the program to include high-level debugging information, and use the debugger to examine the code. Compile the program using:

```
armcc -g hello.c -o hello2
```

where the `-g` option instructs the compiler to add debug information.

Load `hello2` into `armsd`:

```
armsd hello2
```

and set a breakpoint on the first statement in `main()` by entering:

```
break main
```

at the `armsd:` prompt.

To execute the program up to the breakpoint, enter:

```
go
```

The debugger reports that it has stopped at breakpoint #1, and displays the source line. To view the ARM registers, enter:

```
reg
```

To list the C source, enter:

```
type
```

This displays the whole source file. `type` can also display sections of code. For example, if you enter:

```
type 1,6
```

lines 1 to 6 of the source will be displayed.

To show the assembly code rather than the C source, type:

```
list
```

# Command-Line Development

---

The assembly around the current position in the program is shown. You can also list memory at a given address:

```
list 0x8080
```

For further information on using the debugger, see **4.3 *armsd*** on page 4-7 or the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

## 4.2.3 Separating the compile and link stages

To separate the compile and link stages, use the `-c` option when running `armcc`. Quit the debugger and then type the following:

```
armcc -c hello.c
```

This will produce the object file `hello.o`, but no executable. To link the object file with a library, and so generate an executable program, issue the command:

```
armlink hello.o libpath/armlib.32l -o hello3
```

replacing `libpath` with the pathname of the toolkit's `lib` directory on your system. The `armlib.32l` file is the version of the library that uses the 32-bit ARM instruction set and runs in a little-endian memory model.

Run the program:

```
armsd hello3
```

`hello3` contains no C source because `hello.o` was compiled without the `-g` option, so attempting to view the source statements with the `type` command will fail. However, it is still possible to reference program locations and set breakpoints on them using the `@` character to reference the low-level symbols.

For example, to set a breakpoint on the first location in `main()`, type:

```
break @main
```

## 4.2.4 Generating assembly language from C

The compiler can also generate assembly language from C. Quit the debugger and enter:

```
armcc -S hello.c
```

at the system prompt.

The `-S` flag instructs `armcc` to write out an assembly language listing of the instructions that would usually be compiled into executable code. By default, the output file will have the same name as the C source file, but with the extension `.s`.

To view the assembly language that was output by `armcc`, display the file `hello.s` on screen using the appropriate operating system command, or load it into a text editor. You should see the following:

```
; generated by Norcroft ARM C vsn 4.xx (Advanced RISC Machines) [Dec
01 1996]
```

```
        AREA |C$$code|, CODE, READONLY
        |x$codeseg| DATA

main
        MOV     ip,sp
        STMDB   sp!,{fp,ip,lr,pc}
        SUB     fp,ip,#4
        CMP     sp,sl
        BLMI    __rt_stkovf_split_small
        ADD     a1,pc,#L000024-.-8
        BL      _printf
        MOV     a1,#0
        LDMDb   fp,{fp,sp,pc}
L000024
        DCB     0x48,0x65,0x6c,0x6c
        DCB     0x6f,0x20,0x77,0x6f
        DCB     0x72,0x6c,0x64,0x0a
        DCB     00,00,00,00

        AREA |C$$data|, DATA

        |x$dataseg|
```

# Command-Line Development

---

```
EXPORT main

IMPORT _printf
IMPORT __rt_stkovf_split_small

END
```

**Note** *Your code may differ slightly from the above, depending on the version of armcc you are using.*

## 4.2.5 For more information

For a description of the ARM C compiler options and the ARM linker options, see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041)

## 4.3 armsd

The ARM command-line debugger, `armsd`, enables you to debug your ARM targeted image using any of the debugging systems described in **3.3 Debugging Systems** on page 3-3.

This section describes how to carry out basic tasks such as loading a C language based image into `armsd` and setting simple breakpoints. For more detailed instructions on how to use `armsd` please refer to the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) which contains a complete description of the `armsd` command-line options.

### 4.3.1 Starting armsd and loading an image

To start `armsd` and load the image you wish to debug, enter the command:

```
armsd {options} imagename {arguments}
```

You can specify:

- any `armsd` options before the image name
- any arguments for the image after the image name

Then use the `armsd` command line to debug your target.

Complex configuration of the ARMulator requires editing of the `armul.cnf`. For further details see **Chapter 5, The ARMulator**.

If you regularly issue the same set of `armsd` commands, you can run these automatically by typing them in a text file called `armsd.ini` which must be in the current directory or the directory specified by the environment variable `HOME`. The commands will be run whenever you start `armsd`.

### 4.3.2 Obtaining help on the armsd commands

To display a list of all the `armsd` commands available, type:

```
help
```

and to display help on a particular command, type:

```
help command_name
```

at the `armsd` command line.

`help` may be abbreviated to `h`.

### 4.3.3 Setting and removing simple breakpoints

A breakpoint halts the image at a specified location.

To set a simple breakpoint on the first statement of a procedure, enter:

```
break procedure_name
```

You can also use the `break` command to set breakpoints:

- on the final statement of a procedure

# Command-Line Development

---

- on the statement specified by its line number in code

To list all the current breakpoints and their corresponding numbers, enter `break` without any arguments.

To remove a breakpoint enter:

<code>unbreak</code>	if only one breakpoint is set
<code>unbreak #n</code>	to delete breakpoint numbered <i>n</i>
<code>unbreak procedure_name</code>	to delete a breakpoint on first statement of procedure <i>procedure_name</i>

You may use any of these methods to remove a breakpoint, regardless of the way in which the breakpoint was set.

`break` may be abbreviated to `b`, and `unbreak` may be abbreviated to `unb`.

## 4.3.4 Setting and removing simple watchpoints

A watchpoint halts the image when a specified register or variable changes.

To set a simple watchpoint on a variable, enter:

```
watch variable
```

To list all the current watchpoints and their corresponding numbers enter `watch` without any arguments.

To remove a watchpoint enter:

<code>unwatch</code>	if only one watchpoint is set
<code>unwatch #n</code>	to delete watchpoint numbered <i>n</i>
<code>unwatch variable</code>	to delete a watchpoint on a specified variable

You may use any of these methods to remove a watchpoint, regardless of the way in which it was set.

`watch` may be abbreviated to `w`, and `unwatch` may be abbreviated to `unw`.

## 4.3.5 Executing the program

When you have loaded (or reloaded) an image you can execute it by entering the command:

```
go
```

Execution continues until:

- a breakpoint halts the image
- a watchpoint halts the image

To stop the execution of an image, press Ctrl-C.

There are two methods to restart an image that is already load:

- reload the target name:  
`reload targetname`  
and then execute the image again:  
`go`
- The second way of restarting an image is to enter:  
`PC = start_address` (typically 0x8000)  
`CPSR = %IFt_SVC32`  
and then type:  
`go`

See **7.10.1 Faking a reset** on page 7-24 for more information.

You can configure your target to run with command-line arguments by entering:

```
let $cmdline = arguments
```

For example:

```
let $cmdline = "-high -p -M"
```

These arguments replace any arguments set when armsd was started.

`go` may be abbreviated to `g`, and `reload` may be abbreviated to `rel`. There is no abbreviation for `let`.

## 4.3.6 Stepping through the program

You can step through your target using the following three commands:

<code>step</code>	executes a single source code line.
<code>step in</code>	steps into the procedural call.
<code>step out</code>	steps out of a function to the line of originating code which immediately follows that function. This command is useful if <code>step in</code> has been used too often.

Enter `where` to display your current position in the target.

`step` may be abbreviated to `s`, and `where` may be abbreviated to `wh`.

## 4.3.7 Exiting the debugger

To unload the target and exit the debugger type:

```
quit
```

You are returned to the command line.

`quit` may be abbreviated to `q`.

# Command-Line Development

---

## 4.3.8 Viewing and setting image variables

To list all the variables defined within the current context, enter:

```
symbols
```

To view the contents of a variable enter:

```
print variable
```

To view type and context information about a variable enter:

```
variable variable
```

To set the value of a variable use the command:

```
let variable = expression
```

`symbols` may be abbreviated to `sy`, `print` may be abbreviated to `p`, and `variable` may be abbreviated to `v`.

## 4.3.9 Displaying source code

To display type code around the current line enter:

```
type
```

If you wish to display assembly code rather than C source, enter:

```
list
```

`type` may be abbreviated to `t`, and `list` may be abbreviated to `l`.

## 4.3.10 Viewing and setting debugger variables

Some features of `armsd` are specified by the value of the debugger variables, which may be viewed and set in the same way as image variables).

For example, the read-write variable `$list_lines` is an integer value which specifies the number of lines displayed when the `list` command is issued.

Note that some `armsd` variables are read-only.



# 5

## The ARMulator

5.1	What is the ARMulator?	5-2
5.2	Models	5-2
5.3	Rebuilding the ARMulator	5-4
5.4	ARMulator facilities	5-6
5.5	Supplied Models: Angel	5-9
5.6	Supplied Models: Dummy MMU	5-11
5.7	Supplied Models: Profiler	5-12
5.8	Supplied Models: Tracer	5-13
5.9	Supplied Models: Watchpoints	5-15
5.10	Supplied Models: Windows Hourglass	5-15
5.11	Supplied Models: Page Table Manager	5-15
5.12	An Example Memory Model	5-17

# The ARMulator

---

## 5.1 What is the ARMulator?

The ARMulator is a collection of programs that emulate the instruction sets and architecture of various ARM processors. It provides an environment for the development of ARM-targeted software on your workstation or PC.

The ARMulator is instruction-accurate. It models the instruction set without regard to the precise timing characteristics of the processor. As a result, it is well suited to software development and benchmarking of ARM-targeted software, though its performance is somewhat slow compared to real hardware. ARMulator also supports a full ANSI C library to allow complete C programs to run on the emulated system.

ARMulator is transparently connected to armsd to provide a hardware-independent ARM software development environment. Communication takes place via the *Remote Debug Interface (RDI)*.

You can supply models written in C or C++ that interface to the ARMulator's external interface.

## 5.2 Models

You can add extra models to ARMulator without altering the existing model. Each model is entirely self-contained, and 'talks' to the ARMulator through a set of defined interfaces. The full definition of these interfaces is in the ARMulator chapter of the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

The source of a number of example models is provided with the ARMulator.

### Basic models

<code>tracer.c</code>	The tracer module can trace instruction execution and events from inside the ARMulator.
<code>profiler.c</code>	Provides the profiling functionality for the ARMulator, both basic instruction sampling and more advanced use, such as profiling cache misses.
<code>winglass.c</code>	Used only with the ARM Debugger for Windows.
<code>pagetab.c</code>	Sets up the MMU/Cache and associated pagetables inside the ARMulator on reset.

### Memory models

<code>armflat.c</code>	A simple memory model that implements a flat model of 4GB RAM.
<code>armmap.c</code>	Another simple memory model, but one that allows you to have an <code>armsd.map</code> file specifying memory layout. (This necessarily slows down emulation speed, so when no <code>armsd.map</code> file is present, ARMulator will use the faster <code>armflat.c</code> model in preference.)

<code>bytelane.c</code>	An example of a memory model 'veneer', ie. a model that sits between the processor and the real memory model. This model converts the accesses from the core into 'byte-lane' (also known as 'byte-strobe') accesses.
<code>trickbox.c</code>	A memory model of a system where accessing various addresses causes aborts, interrupts, and so on to occur.
<code>tracer.c</code>	As well as being a basic model, the tracer module provides a veneer memory model which can log memory accesses.
<code>armpie.c</code>	(Unix only.) A model of the ARM PIE card.
<code>ebsa110.c</code>	(Unix only.) A model of Digital Semiconductor's EBSA-110 StrongARM evaluation board.
<code>example.c</code>	This memory model is the example described in <b>5.12 An Example Memory Model</b> on page 5-17.

## Coprocessor models

<code>dummysmmu.c</code>	A cut-down model of coprocessor 15 (the system coprocessor).
<code>validate.c</code>	A small coprocessor that can cause interrupts, busy-waits, and so on, used to validate the behavior of the ARM emulator and supplied as an example.

## Operating System models

<code>angel.c</code>	An implementation of the <i>Software Interrupts (SWIs)</i> and environment required for running programs linked with either the Angel or Demon semihosted C libraries on ARMulator.
<code>validate.c</code>	A cut-down SWI handler for running software that primarily deals with SWIs itself. (This model also instantiates the <code>validate</code> coprocessor.)
<code>noos.c</code>	A dummy operating system model, in which no SWIs are intercepted.

Each of these models exports a stub (see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041)). You declare stubs in `models.h`, using sets of macros:

```
MEMORY(ARMul_Flat)
COPROCESSOR(ARMul_DummyMMU)
OSMODEL(ARMul_Angel)
MODEL(ARMul_Profiler)
```

There are no trailing semicolons on these lines. You should also add new user-supplied models to `models.h`.

# The ARMulator

---

## 5.3 Rebuilding the ARMulator

`example.c` defines an extra memory model (see **5.12 An Example Memory Model** on page 5-17). For ARMulator to know about this model, you must declare the model in `models.h` by adding the line:

```
MEMORY(ExampleMemory)
```

You must also add the object file to the supplied Makefile, along with a rule for building the model.

### 5.3.1 Rebuilding on Unix

- 1 Place the source code in the directory `sources`.
- 2 Load the Makefile in `build/` into an editor.
- 3 Add the object to the list of objects to be built.

Change the lines:

```
USEROBSJS = \  
validate.o dummymmu.o angelo noos.o pidprint.o tracer.o \  
armflat.o armpie.o bytelane.o trickbox.o profiler.o models.o \  
winglass.o errors.o armmmap.o watchpnt.o pagetab.o
```

to read:

```
USEROBSJS = \  
validate.o dummymmu.o angelo noos.o pidprint.o tracer.o \  
armflat.o armpie.o bytelane.o trickbox.o profiler.o models.o \  
winglass.o errors.o armmmap.o watchpnt.o pagetab.o example.o
```

- 4 Add a rule for building the example

```
example.o: $(SRC)example.c  
$(CC) $(CFLAGS) -o $@ -c $(SRC)example.c
```

- 5 In directory `build`, type `make`

For the SunOS/gcc target, this produces the output:

```
gcc -g -O2 -DSEEK_SET=0 -DSEEK_CUR=1 -DSEEK_END=2 -o example.o  
-c ../source/example.c  
ranlib ../obj/sarmul.a ../obj/iarm.a ../obj/armulib.a ../obj/  
asdlb.a ../obj/armdbg.a ../obj/clx.a  
gcc -g -O2 -DSEEK_SET=0 -DSEEK_CUR=1 -DSEEK_END=2 hostos.o  
drivers.o serd rive.o spdrive.o validate.o dummymmu.o angelo  
noos.o pidprint.o tracer.o armflat.o armpie.o bytelane.o  
trickbox.o profiler.o models.o winglass.o errors.o armmmap.o  
watchpnt.o pagetab.o example.o ../obj/armstd.o ../obj/pisd.o ../  
obj/angsd.o ../obj/sarmul.a ../obj/iarm.a ../obj/armulib.a ../  
obj/asdlb.a ../obj/armdbg.a ../obj/clx.a -o armstd -lm  
$
```

## 5.3.2 Rebuilding on Windows

To rebuild the ARMulator, load `armulate.mak` into Microsoft Visual C++ Developer Studio (version 4.0 or greater).

Alternatively, type `nmake armulate.mak`.

# The ARMulator

---

## 5.4 ARMulator facilities

### 5.4.1 Controlling ARMulator using the debugger

The debugger (whether it is the command-line `armsd` or the ARM Debugger for Windows) talks to ARMulator over the RDI. Therefore, the level of control is limited by the expressiveness of the RDI.

The RDI allows the debugger to configure:

- processor type
- clock speed

Only one clock speed is allowed, usually taken to be the processor clock speed. For systems with multiple clocks (for example, a cached processor), the clock speed is set in the configuration file (see **5.4.2 Using the `armul.cnf` configuration file** on page 5-6). A default clock speed is taken from this file.

- memory map

The debugger reads the `armsd.map` file and then tells ARMulator of its contents. Individual memory models have to support this information if they are to use the `armsd.map` file.

One such model, `armmap.c`, is supplied with the ARMulator as an example.

Other information is sent over the RDI. Models can intercept the `UnkRDIInfoUpcall` to receive this data. Some of the example models do this. For example:

<code>armmap.c</code>	intercepts the memory map information coming from the debugger. See <b>5.4.4 <code>armsd.map</code> files</b> on page 5-8.
<code>angel.c</code>	intercepts <code>RDIErrorP</code> , <code>RDISet_Cmdline</code> and <code>RDIVector_Catch</code> .
<code>dummymmu.c</code>	responds to the debugger's request about the emulated MMU.
<code>profiler.c</code>	intercepts the profiling calls from the debugger to set up profiling maps, enable profiling, write-back profiling data, and so on.
<code>watchpnt.c</code>	responds to the <code>RDIInfo_Points</code> call from the debugger, responding that watchpoints are available.

**Note** *It is not possible, in this release, to add further control of the ARMulator from the debugger by, for example, the addition of extra commands or pseudo-variables.*

### 5.4.2 Using the `armul.cnf` configuration file

The `armul.cnf` file contains the configuration for the ARMulator. It sets the options for the various ARMulator components, describes different processors, caches, and so on.

**Note** *The format of `armul.cnf` as described below may change in future releases.*

In this release, `armul.cnf` consists of entries of the form `TAG=VALUE` and sections of the form:

```
{ SectionName
  Tag=Value
  Tag=Value
}
```

or

```
SectionName:Tag=Value
```

Models look up values using the `ToolConf` interface.

Each model can have a configuration associated with it in the `armul.cnf` file. The supplied models which have such a configuration are described later in this chapter.

### 5.4.3 Configuring ARMulator to use the example

To tell ARMulator to use the model, you edit the configuration file. The ARMulator reads the configuration file at runtime, so you can switch models without recompiling.

By default, the ARMulator still uses the built-in `Flat` or `MapFile` memory models. To change this behaviour, edit the ARMulator configuration file `armul.cnf` as follows.

Load the `armul.cnf` file into a text editor, and find the lines:

```
...
;; List of memory models
{ Memories
...
;; the 'default' default is the flat memory map
Default=Flat
...

```

about halfway through the file. Change them to:

```
...
;; List of memory models
{ Memories
...
;; Use our new memory model instead
Default=Example
...

```

where `Example` is the name of the model in the `MemStub`, above.

The line you have added states that the default memory model is now `Example`, rather than `Flat` or `MapFile`.

# The ARMulator

---

When you start the ARMulator by running `armsd` or the ARM Debugger for Windows, it responds:

```
ARMulator 2.0
```

```
ARM7, User manual example, 1Mb memory, Dummy MMU, Soft Angel 1.4  
[Angel SWIs, Demon SWIs], FPE initialisation failed, Profiler,  
Tracer, Pagetables, Big endian.
```

The *Floating Point Emulator (FPE)* initialization failed because this model does not have a standard memory map, and the FPE could not be loaded.

Alternatively, you might see the error: `Initialisation failed: Memory model 'Example' incompatible with bus interface`. This is the memory model reporting that it cannot talk to the selected processor (for example, StrongARM, ARM7TDMI, or ARM8).

## 5.4.4 armsd.map files

The format of the `armsd.map` file is described in the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

It is the responsibility of the memory model to implement map files. New models do not use the map file unless support is written in. Only one supplied model, `armmap.c`, supports it.

To support the map data, a memory model has to intercept upcall `UnkRDIInfoUpcall`, watching for:

- `RDIMemory_Map`

The debugger makes this call to pass the data parsed from the `armsd.map` file.

- `arg1` points to an array of `RDI_MemDescr` structures.
- `arg2` gives the number of elements in the array.

`RDIMemory_Map` may be called a multiple number of times.

- `RDIInfo_Memory_Stats`

The model should return `RDIError_NOError` to indicate that memory maps are supported.

- `RDIMemory_Access`

The debugger makes this call to obtain access statistics (see `$memory_statistics` or the equivalent in the ARM Debugger for Windows).

- `arg1` points to an `RDI_MemAccessStats` structure for the memory model to fill in. (One call is made for each mapped area passed to `RDIMemory_Map`.)
- `arg2` identifies the area by the handle passed in the `RDI_MemDescr` passed to `RDIMemory_Map`.

These structures are defined in `dbg_stat.h`. For full details of these RDI calls, refer to the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).



## 5.5 Supplied Models: Angel

The Angel model (`angel.c`) is an operating-system model that allows code that has been built to run with the Angel Debug Monitor to run under ARMulator. The Angel model also provides backwards compatibility with the Demon Debug Monitor.

The model intercepts the Angel SWI, and emulates the functionality of Angel directly on the host, transparently to the program running on the ARMulator.

### 5.5.1 Configuring Angel

The configuration for the Angel model exists in a section called `OS` in the `armul.cnf` file. This appears as:

```
{ OS
;; Angel configuration
[ ...]
}
```

The configuration options are:

```
AngelSWIARM=0x123456
AngelSWIThumb=0xdfab
```

`AngelSWIARM` and `AngelSWIThumb` declare the SWI numbers that Angel uses. For descriptions, see the Angel chapter in the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

```
HeapBase=0x02069000
HeapLimit=0x02079000
StackBase=0x02080000
StackLimit=0x02079000
```

`HeapBase` and `HeapLimit` define the application heap; `StackBase` and `StackLimit` define the application stack.

```
Demon=Yes
```

`Demon` declares whether backwards compatibility with the Demon Debug Monitor is provided. If not, the FPE cannot be loaded.

The Angel model automatically detects at runtime whether a model uses Angel or Demon SWIs.

### 5.5.2 Configuring Demon

The Demon configuration options appear alongside the Angel options in `armul.cnf`:

```
AddrSuperStack=0xa00
AddrAbortStack=0x800
AddrUndefStack=0x700
```

# The ARMulator

---

```
AddrIRQStack=0x500
```

```
AddrFIQStack=0x400
```

The options above define the initial locations of the exception mode stack pointers. These apply equally to Angel and Demon.

```
AddrUserStack=0x80000
```

This option is the default location of the user-mode stack, and the default value returned by `SWI_GetEnv` (which returns the top of application memory). Some other value may be returned if, for example, a memory model calls `ARMul_SetMemSize`.

```
AddrSoftVectors=0xa40
```

```
AddrsofHandlers=0xad0
```

```
SoftVectorCode=0xb80
```

These options define where in memory the ARMulator places the hardware exception vector handling code.

```
AddrCmdLine=0xf00
```

This option points to a buffer where the Angel model places a copy of the command line (retrieved by catching the `RDISet_Cmdline RDI_info` call).

## 5.6 Supplied Models: Dummy MMU

DummyMMU (`dummymmu.c`) provides a dummy implementation of an ARM Architecture v.3/v.4 system coprocessor. This does not provide any of the cache and MMU functions, but does prevent accesses to this coprocessor being Undefined Instruction exceptions.

Reads from register 0 return a dummy ARM ID register value, which may be configured.

Writes to register 1 of the dummy coprocessor (the system configuration register) set the **bigend**, **lateabt** and other signals.

### 5.6.1 Configuring the Dummy MMU

The DummyMMU may have its ID code set in the configuration file. Use the following entry in the `Coprocessors` section of `armul.cnf`:

```
{ DummyMMU
;; The Dummy MMU can be configured to return a given Chip ID
;DummyMMU:ChipID=
}
```

This parameter can be uncommented and set to any value. For example, to configure DummyMMU to return the ARM710 ID code (`0x44007100`), change these lines to:

```
{ DummyMMU
;; The Dummy MMU can be configured to return a given Chip ID
ChipID=0x44007100
}
```

# The ARMulator

---

## 5.7 Supplied Models: Profiler

`profiler.c` contains code to implement the profiling options in the debugger. It does so by providing an `UnkRDIInfoHandler` which handles the profiling requests from the debugger.

In addition to profiling program execution time, it allows you to use the profiling mechanism to profile events, such as cache misses. The `Profiler` section in the configuration file controls this.

```
{ Profiler
;; For example - to profile cache misses set:
;Type=Event
;Event=0x00010001
}
```

By default, this is empty. If uncommented, the example shown will allow profiling of cache misses.

The Profiler also allows `Type=Time` (the default is that samples are taken every *n* microseconds) and `Type=Instruction` (samples are taken every *n* instructions). When `Type=Event` the profiling interval is ignored. `EventMask` is also allowed (see the section **5.8 Supplied Models: Tracer** on page 5-13).

### 5.7.1 Using the Profiler

The Profiler is controlled by the debugger. See the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for details.

## 5.8 Supplied Models: Tracer

An example implementation of a tracer is provided. This can trace instructions, memory accesses, and events to a file (in text or binary format). See the source, `tracer.c`, for details of the formats used in these files. The configuration file `armul.cnf` controls what is traced.

Alternatively, you can link your own tracing code onto the Tracer module, allowing real-time tracing. No examples are supplied, but the three required functions are documented here.

```
unsigned Tracer_Open(Trace_State *ts)
```

This is called when the tracer is initialized. The implementation in `tracer.c` opens the output file from this function, and writes a header.

```
void Tracer_Dispatch(Trace_State *ts, Trace_Packet *packet)
```

This is called on each traced event for every instruction, event, or memory access. In `tracer.c` this function writes the packet to the trace file.

```
void Tracer_Close(Trace_State *ts)
```

This is called at the end of tracing. `tracer.c` uses this to close the trace file.

```
extern void Tracer_flush(Trace_state *ts)
```

This is called when tracing is disabled. `tracer.c` uses this to flush output to the trace file.

The formats of `Trace_State` and `Trace_Packet` are documented in `tracer.h`.

The default implementations of these functions can be changed by compiling `tracer.c` with `EXTERNAL_DISPATCH` defined.

### 5.8.1 Configuring the Tracer

The Tracer has its own section in the ARMulator configuration file (`armul.cnf`).

The Tracer can be used either as a memory model (in which case it can trace memory accesses) or as a basic model (which can only trace instructions or events).

To use the Tracer as a memory model, you must configure it for the processor being emulated. To make this process easier, the configuration file contains the following entry near the top:

```
;; Controls whether there's a trace on memory accesses.
;TraceMemory
```

Uncomment this by removing the semicolon before `TraceMemory`. Then, to configure the tracer, find the `Memories` section in the configuration file, and the `Tracer` section below it. This appears as follows:

```
{ Tracer
;; Tracer options
File=armultrc
TraceInstructions
```

# The ARMuLator

---

```
;TraceMemory
;TraceIdle
;TraceEvents
;Disassemble
```

`File` defines the file to which the trace information will be dumped, using the default `Tracer_Open` functions. Alternatively, you can use `BinFile` to store data in a binary format. The other four options (three of which are, by default, commented out) control what is being traced. `TraceMemory` traces only real memory accesses, not idle cycles, so there is also a `TraceIdle` option.

The configuration for the Tracer as a basic model is further down the configuration file, in another subsection called `Tracer` in the `Models` section. The configuration is the same, except that `TraceMemory` and `TraceIdle` are ignored.

You can also control tracing using:

- `Range=low address,high address`  
Only things in the specified address range are traced.
- `Sample=n`  
Only every *n*th traced event is sent to the trace file.
- `Disassemble`  
Instructions are disassembled. This greatly affects performance.

When tracing events, you can select the events to be traced using:

- `EventMask=mask,value`  
Only those events whose number when masked (bitwise-AND) with *mask* equals *value* are traced.
- `Event=number`  
Only *number* is traced. (This is equivalent to `EventMask=0xffffffff,number`.)

## 5.8.2 Using the Tracer

There is no direct debugger support for tracing, so instead the tracer uses bit 4 of the RDI Logging Level (`$rdi_log`) variable.

### Using the ARM Debugger for Windows (ADW)

Select **Set RDI Log Level** from the **Options** menu. To enable tracing, set the RDI Log Level to 16. To disable tracing, set the RDI Log Level to 0.

See **3.4.9 Remote Debugging Interface** on page 3-9 for more information.

## Using armsd (Unix):

To enable tracing, type:

```
armsd: $rdi_log=16
```

To disable tracing, type:

```
armsd: $rdi_log=0
```

## 5.9 Supplied Models: Watchpoints

The Watchpoints module is a memory veneer that provides memory watchpoints. It sits between the processor core and memory (or cache, as appropriate).

If you enable the Watchpoints module, performance is improved when you are using watchpoints, but overall performance is lowered. To enable watchpoints, uncomment the `Watchpoints` line:

```
;; To enable watchpoints, set "WatchPoints"
; Watchpoints
in armul.cnf.
```

## 5.10 Supplied Models: Windows Hourglass

This module deals with calling the debugger regularly during execution. This is required when you are using the ADW. The `WindowsHourglass` section in the configuration file controls how regularly this occurs:

```
{ WindowsHourglass
Rate=8192
}
```

Increasing this rate decreases the regularity at which control is yielded to ADW. This increases emulation speed but decreases responsiveness.

## 5.11 Supplied Models: Page Table Manager

The PageTab module is a simple model that sets up pagetables and initializes the MMU on reset. The `Pagetables` section in the configuration file controls the contents of the pagetables, and the configuration of the MMU:

```
{ Pagetables
```

The first set of flags controls the MMU and cache. See the *ARM Architecture Reference Manual* (ARM DDI 0100) for details of these flags.

Some only apply to certain processors. For example, `BranchPredict` only applies to the ARM810, and `ICache` to the SA-110 processor.

```
MMU=Yes
AlignFaults=No
Cache=Yes
```

# The ARMulator

---

```
WriteBuffer=Yes
Prog32=Yes
Data32=Yes
LateAbort=Yes
BigEnd=No
BranchPredict=Yes
ICache=Yes
```

The second set of options control the Translation Table Base Register (System Control Register 2) and the Domain Access Control Register (Register 3). Again, see the *ARM Architecture Reference Manual* (ARM DDI 0100) for details. The Translation Table Base Register should be aligned to a 16kB boundary.

```
PageTableBase=0xa0000000
DAC=0x00000003
```

Finally, the configuration file can contain a basic outline of the pagetable contents. The module will write out a top-level pagetable (to the address specified for the Translation Table Base Register) whenever ARMulator resets.

Regions are named `Region[n]`, where *n* is an integer, starting with `Region[0]`:

```
{ Region[0]
VirtualBase=0
PhysicalBase=0
Pages=4096
Cacheable=Yes
Bufferable=Yes
Updateable=Yes
Domain=0
AccessPermissions=3
Translate=Yes
}
```

The `VirtualBase` is the virtual address of the base of this region; `PhysicalBase` is the address that it will map to. (If `PhysicalBase` is not specified, it defaults to be the same as `VirtualBase`.) Both addresses should be aligned to a 1MB boundary.

`Pages` specifies the number of 1MB pages that this region covers.

`Cacheable`, `Bufferable` and `Updateable` control the C, B and U bits in the translation table entry. (See the *ARM Architecture Reference Manual* (ARM DDI 0100) for details. Note that the U bit is only used for the ARM610 processor.) Similarly `Domain` specifies the domain field of the table entry, and `AccessPermissions` the AP field.

Finally, `Translate` controls whether accesses to this region will cause translation faults. Setting `Translate=No` for a region will cause an abort to occur whenever ARMulator reads or writes to that region.



5.12 An Example Memory Model

The example memory model includes a simple address decoder, a memory mapped I/O area, and some RAM that is paged by writing to another area of memory.

5.12.1 The memory map

The example memory model supplied with ARMulator is in `armflat.c`. This models just a flat 4GB of RAM.

This example deals with `example.c`, a device in which memory is split into two 128KB pages. The bottom page is read-only, and the top page has one of eight 128KB memory pages mapped into it, page 0 being the 'low page'. Such a system might be used to implement a small number of user tasks.

Addresses wrap around above 256KB for the first 1GB of memory, as if bits 29:18 of the address bus are ignored. Bits 31:30 are statically decoded:

bit 31	bit 30	
0	0	Memory access.
0	1	Bits 18:16 of the address select the physical page mapped in to the top page.
1	0	I/O port. (see below)
1	1	Generates an abort.

Table 5-1: Address bus

This produces the memory map shown in *Figure 5-1: Memory map* on page 5-18.

# The ARMulator

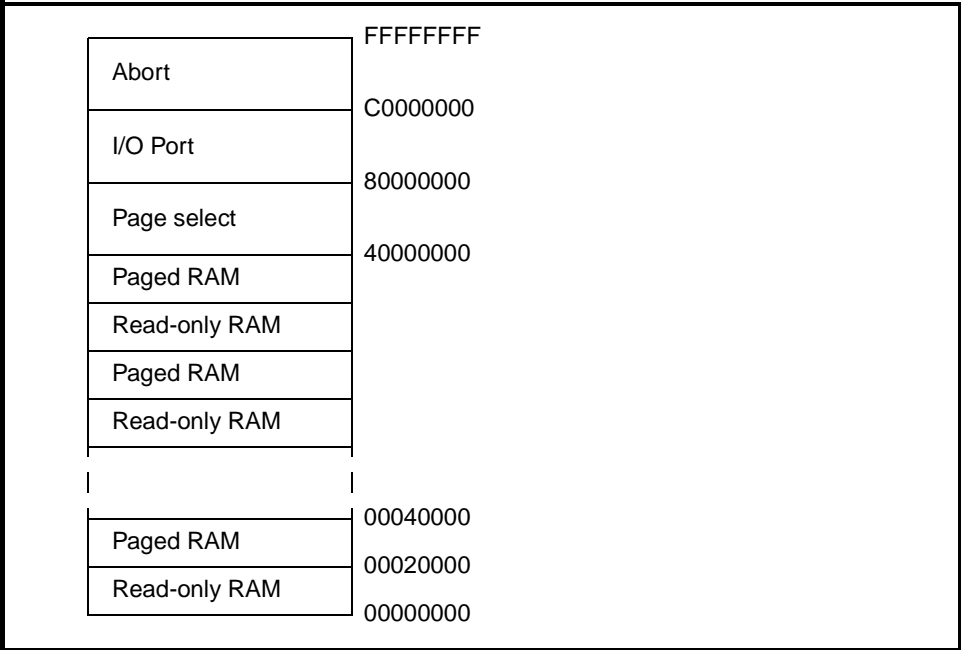


Figure 5-1: Memory map

The I/O area, which is accessible only in ‘privileged’ modes, is split as follows:

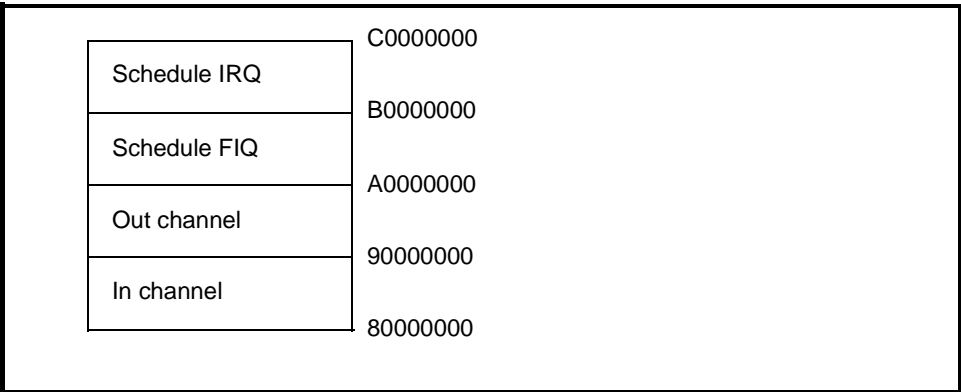


Figure 5-2: I/O area split



These function as follows:

<b>Schedule FIQ</b>	An FIQ is raised after $n$ cycles, where $n$ is the bottom 8 bits of the address.
<b>Schedule IRQ</b>	An IRQ is raised after $n$ cycles, where $n$ is the bottom 8 bits of the address.
<b>Out channel</b>	The character represented by the bottom 8 bits of the data is sent to the screen for a write, and is ignored on read.
<b>In channel</b>	A byte is read from the terminal for a read, or ignored for a write.

## 5.12.2 Implementation

There are eight banks of 128KB of RAM, one of which is currently mapped in to the top page. The memory model has two *pieces of state*: an array representing the model of memory, and the number of the page currently mapped into the 'top' page. As a result, a very simple data structure is used:

```
#include <string.h>           /* for memset */
#include "armdefs.h"
#include "dbg_hif.h"          /* so we can use the HostosInterface */
#define PAGE_SIZE (1<<17)
typedef union {
    char byte[PAGE_SIZE];
    ARMword word[PAGE_SIZE/4];
} page;

typedef struct {
    page *p[8];               /* eight pages of memory */
    int mapped_in;
    int Ntrans;
    ARMul_State *state;       /* So we can generate interrupts */
    const Dbg_HostosInterface *hif; /* So we can output characters */
    int fiq_cnt, irq_cnt;     /* Counters for interrupts */
} ModelState;
```

In this model the ARM does not need to run in different endian modes. You can assume that the ARM is configured to be the same endianness as the host architecture.

**Note** *If you want to allow the ARM to run in different endian modes, you must have a ConfigChange callback, as in armflat.c.*

# The ARMulator

---

However, you do occasionally need to ensure that a write is allowed only if the **Ntrans** signal is HIGH, indicating that the processor is in a privileged mode. To enable you to know this, you must install a callback for changes to **Ntrans**, because it is not supplied to the memory access function. The core calls the callback whenever **Ntrans** changes (on mode changes), and when executing an LDRT/STRT instruction.

```
static void TransChangeHandler(void *handle,
    unsigned old,unsigned new)
{
    ModelState *s=(ModelState *)handle;
    s->Ntrans=new;
}
```

You must export a stub for the model:

```
static armul_Error MemInit(ARMul_State *state,
    ARMul_MemInterface *interf,
    ARMul_MemType type,
    toolconf config);

#define ModelName "Example"
ARMul_MemStub ExampleMemory = {
    MemInit,
    ModelName
};
```

The initialization function must do three main things:

- fill in the **MemInterface** structure passed in
- allocate the model's state
- install the **Ntrans** change and exit handlers

```
/*
```

Predeclare the memory access functions so that the initialize function can fill them in

```
*/
```

```
static int MemAccess(void *,ARMword,ARMword *,ARMul_acc);
extern void free(void *); /* ANSI definition of 'free' */
#define OFFSET(addr) ((addr) & 0x7fff)
#define WORDOFF(addr) (OFFSET(addr)>>2)
static armul_Error MemInit(ARMul_State *state,
    ARMul_MemInterface *interf,
    ARMul_MemType type,
    toolconf config)
```

```
{
ModelState *s;
int i;
ARMul_PrettyPrint(state, ", User manual example");
/* don't support ReadClock and ReadCycles */
interf->read_clock=NULL;
interf->read_cycles=NULL;
Only the ARM6/ARM7 memory interfaces are supported, so raise an error if it is not one of
these.
if (type!=ARMul_MemType_Basic && type!=ARMul_MemType_BasicCached) {
return
    ARMul_RaiseError(state,ARMulErr_MemTypeUnhandled,ModelName);
}
```

Now allocate the state:

```
s=(ModelState *)malloc(sizeof(ModelState));
if (s==NULL) return ARMul_RaiseError(state,ARMulErr_OutOfMemory);
and install the function to free it:
```

```
ARMul_InstallExitHandler(state, free, (void *)s);
for (i=0;i<8;i++) {
    s->p[i]=(page *)malloc(sizeof(page));
    if (s->p[i]==NULL) return FALSE;
    memset(s->p[i], 0, sizeof(page));
}
s->mapped_in=0;
s->state=state; /* keep a handle onto the ARMulator state */
s->hif=ARMul_HostIf(state); /* and grab the handle onto the
HostosInterface */
ARMul_PrettyPrint(state, ", 1Mb memory");
```

On a cached processor there is no **Ntrans** signal, so it is always treated it as being HIGH.

```
/* Install the mode change handler */
if (type==ARMul_MemType_BasicCached) {
s->Ntrans=1;
} else {
ARMul_InstallTransChangeHandler(state,
    TransChangeHandler,
    (void *)s);
}
```

# The ARMuLator

---

```
/*
Call SetMemSize so that the debug monitor knows where to place the
stack.
*/
ARMul_SetMemSize(state, 8*PAGESIZE);
interf->x.basic.access=MemAccess;
interf->handle=(void *)s;
return ARMulErr_NoError;
}
```

Finally, write the generic access function:

```
static int MemAccess(void *handle,
                     ARMword address,
                     ARMword *data,
                     ARMul_acc acc)
```

```
{
    int highpage=(address & (1<<17));
    ModelState *s=(ModelState *)handle;
    page *mem;
```

```
/*
Get a pointer to the correct page of memory to read/write from:
*/
```

```
if (highpage)
    mem=s->p[s->mapped_in];
else
    mem=s->p[0];
/*
```

Next, deal with the interrupt counters as follows:

```
*/
if (s->fiq_cnt && --s->fiq_cnt==0)
    ARMul_SetNfiq(t->state, 0);
if (s->irq_cnt && --s->irq_cnt==0)
    ARMul_SetNirq(t->state, 0);
/* acc_MREQ is true if this is a memory cycle */
if (acc_MREQ(acc)) {
    /* Now decode the top-bits of the address */
    switch (address>>30) { /* decode bits 30,31 */
    case 0: /* 00 - memory access */
        if (acc_READ(acc)) {
```

```

        switch (acc_WIDTH(acc)) {
        case BITS_32:
            *data=mem->word[WORDOFF(address)];
            return 1;
        case BITS_8:
            *data=mem->byte[OFFSET(address)];
            return 1;
        default: /* do not understand this request */
            return -1;
        }
    } else { /* write */
/*
Ignore writes out of supervisor modes to the 'low' page, and writes
in svc modes from STRT instructions:
*/
if (highpage || /* not the lowpage */
    !acc_ACCOUNT(acc) || /* or from the debugger */
    s->Ntrans) {
    switch (acc_WIDTH(acc)) {
    case BITS_32: /* word */
        mem->word[WORDOFF(address)]=*data;
        return 1;
    case BITS_8: /* byte */
        mem->word[OFFSET(address)]=*data;
        return 1;
    default: /* should not happen */
        return -1;
    }
}
}
case 1: /* 01 - page select in SVC mode */
    /* Changing the mapped in page is simple enough. */
    if (s->Ntrans || !acc_ACCOUNT(acc)) {
        s->mapped_in=(address>>16) & 7;
    }
    return 1;
case 2: /* 10 - I/O area */

```

# The ARMulator

---

```
/*
Carry out a further decoding, but allow these accesses only from
'privileged' modes:
*/
if (s->Ntrans)
    switch ((address>>28) & 0x3) { /* decode bits 28,29 */
    case 0: /* read byte */
        if (acc_READ(acc))
            *data=s->hif->readc(s->hif->hostosarg);
    case 1:
        if (acc_WRITE(acc))
            s->hif->writec(s->hif->hostosarg,(*data) & 0xff);
        break;
    case 2: /* Schedule IRQ */
        s->irq_cnt=address & 0xff;
        break;
    case 3: /* Schedule FIQ */
        s->fiq_cnt=address & 0xff;
        break;
    }
    return 1;
    case 3: /* 11 - generate an abort */
/*
Unlike previous ARMulators, returning -1 generates an abort, and
it does not matter what kind of abort it is.
*/
    return -1;
}
} else { /* not a memory request */
/*
MemAccess is called for all ARM cycles, not just memory cycles,
and must keep count of these I and C cycles. Therefore, the code
can return 1, just as for a memory cycle, because returning 0
indicates that the memory is stalling the processor.
*/
    return 1;
}
}
```



# 6

## Angel

6.1	Introduction	6-2
6.2	Overview of Developing Applications with Angel	6-4
6.3	Inside Angel	6-9
6.4	Angel System Features	6-12
6.5	Debuggers	6-16
6.6	Minimal Angel	6-17
6.7	Porting Angel to New Hardware	6-18
6.8	Downloading a Debug Agent	6-21
6.9	Adding a Channel	6-22
6.10	Using Angel on the ARM PID7T board	6-23
6.11	Using Angel on the ARM60 PIE Board or ARM7 PIE Board	6-26
6.12	The Debug Comms Channel, Angel, and EmbeddedICE	6-27
6.13	Notes for Demon Users	6-29
6.14	Example of an Application Device using Angel (UNIX only)	6-31

## 6.1 Introduction

This chapter introduces the Angel development and debugging system. To aid readability, the term *Angel* is used to mean *the Angel Debug Monitor* throughout this chapter.

Angel is a program that allows rapid development and debugging of applications running on ARM-based hardware. Angel can debug applications running in both ARM and Thumb state on target hardware.

A typical Angel system has two main components which communicate via a physical link, such as a serial cable:

*Debugger*

This runs on the host computer. It gives instructions to Angel and displays the results obtained from it. The debugger could be armsd, the ARM Debugger for Windows, or any other debugging tool which can handle the communications protocol used by Angel.

*Angel Debug Monitor* This runs alongside the application being debugged on the target platform. There are two versions of Angel:

- a full version for use on development hardware
- a minimal version intended for use on production hardware

Angel uses a debugging protocol called the *Angel Debug Protocol (ADP)*, which supports multiple channels. It is easy to port to different hardware. It requires control over the ARM's exception vectors, but can share these with your applications.

Angel can be used for:

- evaluating existing application software on real hardware (as opposed to hardware emulation)
- developing software applications on development hardware
- bringing into operation new hardware that includes an ARM processor
- porting operating systems to ARM-based hardware

These activities require some understanding of how Angel's components work together. The more technically challenging ones, such as porting operating systems, involve some modification of Angel itself.

This chapter describes aspects of the process of developing applications under Angel. More information about Angel can be found in the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

## 6.1.1 Angel and Demon

Angel is similar in functionality to Demon. Demon is the debug monitor contained in the ARM Software Development Toolkit release 2.0. However, Angel is more modular in design, easier to port, and easier to customize. It is also able to support a number of facilities that were unavailable with Demon.

For more information about the differences between Angel and Demon, see **6.13 Notes for Demon Users** on page 6-29.

For more information about Demon, see *Application Note 39: Demon and RDP* (ARM DAI 0039)

## 6.2 Overview of Developing Applications with Angel

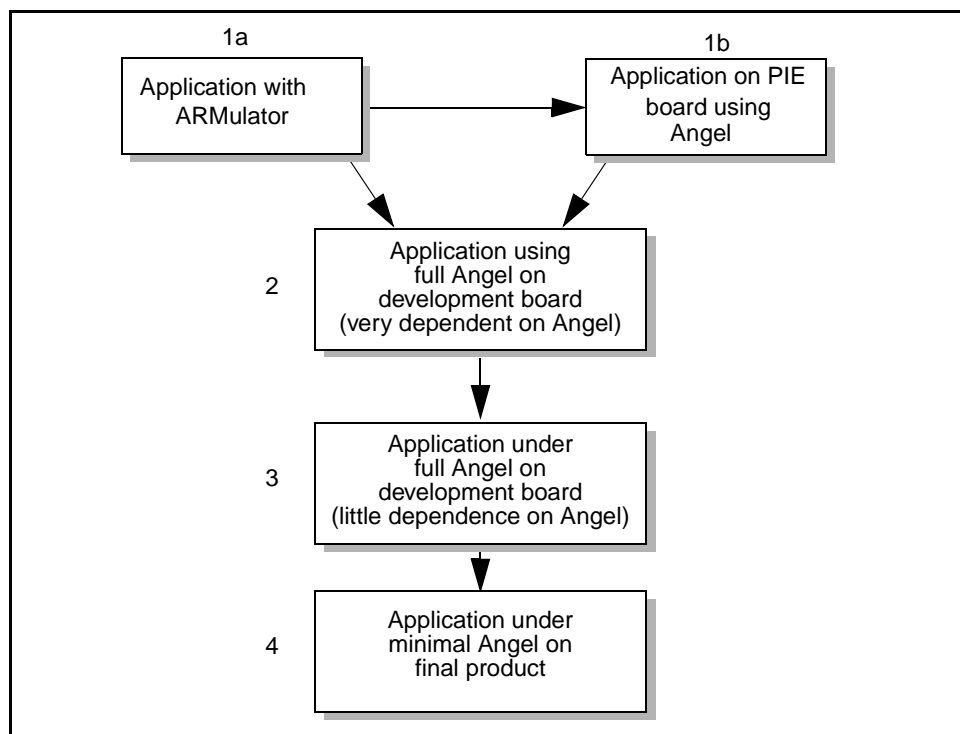
This section gives an overview of the development process of an application using Angel, from the evaluation stage up to the final product.

### 6.2.1 Development cycle

The stages in the Angel development cycle are:

- 1 Evaluate the application.
- 2 Build with high dependence on Angel.
- 3 Build with low dependence on Angel.
- 4 Move to final production hardware.

The details of these changes are shown in **Figure 6-1: The Angel development cycle**.



**Figure 6-1: The Angel development cycle**

## 6.2.2 Stage 1a: Evaluating applications under ARMulator

You are likely to want to evaluate the ARM to check that it is appropriate for your application. To do this, you need a program or suite of programs to run on the ARM.

You recompile your programs using the ARM C compiler, and link them with the ARM semihosted C library. The programs can then be run under the ARMulator. Cycle counts from ARMulator can be evaluated to see if the performance was sufficient.

This stage does not involve Angel, although you use an Angel-compatible ARM C library.

## 6.2.3 Stage 1b: Evaluating applications on PIE board under full Angel

Instead of trying out semihosted programs (ie. ones which make use of the ARM C library) on an ARM Emulator, you could use an ARM PIE board to do an evaluation.

Although this system does involve Angel (running as a debug monitor on the ARM PIE board), there is no need for you to rebuild Angel, or to be familiar with the way Angel works.

At this stage, you would build *ARM Image Format (AIF)* images which do not include an Angel Debug Monitor, and then download these using the ARM debuggers. (You use the same approach with Demon.)

## 6.2.4 Stage 2: Building applications on a development board, highly dependent on Angel

This is conceptually the same as Stage 1, except that the target board is your own, rather than an ARM PIE board. It may, therefore, have different peripheral hardware, different memory maps, and so on.

Because of the different hardware, you must port Angel to your development board. This includes writing device drivers for any devices not supported by Angel on a PIE board.

At this stage, you can build your application in one of two ways:

- Build a stand-alone application (that does not include Angel) which must be downloaded by the debugger.
- Build an application including Angel that can then be blown into a ROM, installed using a ROM emulator or EmbeddedICE, or soft-loaded into RAM by the ARM debuggers. This ROM image can be downloaded by the ARM debuggers. This is conceptually a step closer to the final product (compared with downloading an AIF image via the debugger).

### Techniques

There are a number of techniques you can use to get successive versions of the application onto the development board. Each technique has advantages and disadvantages:

- 1 Using Angel with a serial port.  
This provides slow downloading, but requires only a simple UART on the development board. However, on its own it cannot be used to change the application installed in ROM or flash RAM permanently.

- 2 Using Angel with serial and parallel ports.  
This provides medium speed downloading, but requires a serial and parallel port on the development board. However, on its own it cannot be used to change the application installed in ROM or flash RAM permanently.
- 3 Using Angel with Ethernet connection.  
This provides fast downloading, but requires Ethernet hardware on the development board and a considerable amount of Ethernet support software to run on the development board. However, on its own it cannot be used to change the application installed in ROM or flash RAM permanently.
- 4 Using Angel with ADP over JTAG.  
This provides fairly slow downloading, but requires that the processor being used has a Debug Comms Channel, and also that the user has an EmbeddedICE board with the *ADP over JTAG* software. See section **6.12.2 ADP over JTAG using EmbeddedICE board** on page 6-28. The main advantage is that no peripherals of any kind are required on the development board to support this method. However, on its own it cannot be used to change the application installed in ROM or flash RAM permanently.
- 5 Blowing a new ROM/EPROM each time.  
This provides slow 'downloading', in that it takes a relatively fixed amount of time to go through the cycle: remove old EPROM, blow new EPROM, insert new EPROM. Also, the time taken to erase EPROMs can be a problem.  
However, for extremely large ROM images, where otherwise only a slow download mechanism is available, this might be preferable. Also, once 'downloaded' in this way, the application is permanently available, and doesn't have to be reloaded when the board is switched off.
- 6 Using a ROM Emulator to 'download' a new ROM image.  
This provides medium to fast downloading, depending on the ROM Emulator. It is essential to have access to a ROM Emulator.
- 7 Flash download.  
This provides slow to fast downloading, depending on which type of connection (methods 1 to 4 above) you are using. But it is possible only on boards that support a Flash RAM, and for systems supported by the Flash download program. Currently, only the ARM7T PID is supported. This method has the advantage that, once the flash is set, the application is fixed in memory, even if the board is switched off.
- 8 ROM emulation using EmbeddedICE.  
This provides slow to medium downloading, depending on the way the EmbeddedICE board is connected (1 and 2 above). Load the ROM image into RAM using EmbeddedICE and then force execution from the start of the image. This method also allows EmbeddedICE's debugging facilities to be used to debug the porting of Angel.

Methods 5 to 8 allow only ROM images to be ‘downloaded’, and not the application-only images (eg. AIF images) which are downloaded in an evaluation system. Therefore, if methods 5 to 8 are used, the change from dealing with application images to ROM images has to be made as soon as the development phase starts.

On the other hand, with methods 1 to 4, it is possible to download either an application image or a ROM image. This makes the transition into the development phase easier. The switch between downloading application images and ROM images must still be made before you prepare to move to production hardware, but the transition can be made when everything else is working, which makes it easier.

## 6.2.5 Stage 3: Building applications on a development board, with little dependence on Angel

This is similar to stage 2, but at a later stage in the development. It highlights the fact that you can build only two versions of Angel:

- the full version
- the minimal version (described in Stage 4)

There are no other ‘cut-down’ versions, because you can simply switch off features such as semihosting without building a special cut-down version of Angel.

Once Angel is working on your board, you can develop your application, making use of the debugging functionality that Angel provides.

## 6.2.6 Stage 4: Moving an application to final production hardware

### Preparing to move to production hardware

When you are ready to move the application onto final production hardware, you have a different set of requirements. The production hardware is unlikely to have as many communications links as the development board. At this stage, RAM and ROM are now very scarce resources, so it is not desirable to include any parts of Angel that are not required in the final product. Interrupt handlers for timers may be required in the final product, but debug support code is not. Also, it is unlikely that you are able to communicate with a debugger over a communications channel, because there are no longer UARTs available purely for debugging.

So, to ease the transition from development hardware to production hardware, you link in a ‘Minimal Angel’, rather than a full version. Minimal Angel does not support features such as:

- debugging
- semihosting
- multiple channels on one device

which are provided by ‘Full Angel’. However, it is structured in the same way as Full Angel, so initialization, device drivers (if required), interrupt support, and so on are all dealt with in the same way. Moving from Full Angel to Minimal Angel on the development hardware should be straightforward. See **6.6 Minimal Angel** on page 6-17 for a description of Minimal Angel.

The best way to debug a Minimal Angel system at this stage is to use EmbeddedICE, which does not take up any resource on the target.

## **Moving to production hardware**

The final stage is to move the Minimal Angel system onto Production Hardware. To do this, use EmbeddedICE as described in **6.2.6 Stage 4: Moving an application to final production hardware** on page 6-7.

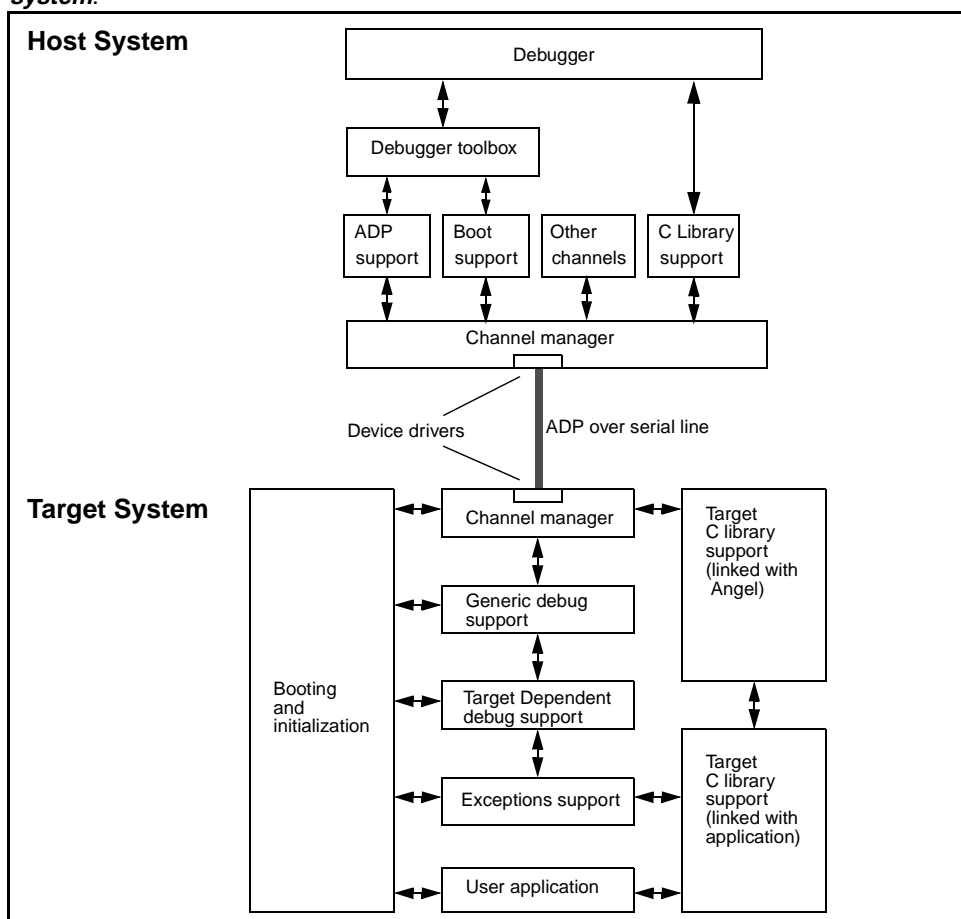


## 6.3 Inside Angel

### 6.3.1 A typical Angel system

A typical Angel system consists of a host machine running a debugger, connected to a board that contains an ARM processor running Angel alongside an application. The host machine sends requests to Angel. Angel interprets those requests at the lowest level and performs an operation such as inserting an undefined instruction where a breakpoint is required or reading a portion of memory and sending back a response to the host.

The main components of an Angel system are shown in **Figure 6-2: A typical Angel system**.



**Figure 6-2: A typical Angel system**

## Host system

Debugger	This is the ARM Debugger for Windows (ADW) or the ARM command-line debugger (armsd).
Debugger toolbox	This provides an interface between the debugger and the Remote Debug Interface (RDI).
ADP support	This translates between host RDI messages and target ADP messages.
Boot support	This establishes communication between the target and host systems, eg. sets baud rates and initializes Angel in the target.
C library support	This handles requests sent by the target C library (semihosting requests).
Host channel manager	This handles the communication channels, providing the functionality of the devices but at a higher level. The channel manager allows you to open multiple channels on a single device, eg. serial port.
Device drivers	These implement particular communications devices on the host. Each driver provides the entry points required by the channel manager.

## Target system

Device drivers	These implement particular communications devices on the development boards. Each driver provides the entry points required by the channel manager.
Channel manager	This handles the communication channels, providing all of the functionality of the devices but at a higher level. It allows you to open multiple channels on a single device, eg. serial port.
Generic debug support	This handles the ADP by communicating with the host over a configured channel and sending and receiving commands from the host. When the system has been debugged, generic debug support can be removed from the system with no other effect.
Target dependent debug support	This provides system-dependent features, eg. setting up breakpoints and reading/writing memory.
Exceptions support	This handles all ARM exceptions including interrupts, Angel internal undefined instructions and any unexpected exceptions.
C library support	This is implemented in two sections, one linked with the application, the other built into Angel to send requests to the host where necessary.

Booting and initialization	This performs start-up checks, sets up memory, stacks, and devices, and sends boot messages to the debuggers.
User application	This is an application on the target system.

## 6.4 Angel System Features

Angel is available in the following forms:

- in stand-alone form as default ROM for PIE and PID boards
- in library form to be linked with user applications

The Angel system provided in the ARM Software Development Toolkit also provides:

- host debuggers which support ADP for Unix, and Windows 95 and NT3.51
- ANSI C Library for use with Angel
- ADP-capable EmbeddedICE software
- ADP over Debug Comms Channel software for EmbeddedICE hardware

### 6.4.1 System resources and configuration

Angel uses as few system resources as possible. For example, it intrudes only minimally on SWI space. Wherever possible, Angel is configurable.

Configurable resources

Angel requires the following for semihosting purposes:

- one ARM SWI
- one Thumb SWI

Non-configurable resources

For breakpoints, Angel also requires:

- two ARM Undefined instructions
- one Thumb Undefined instruction

#### Configuration

Angel is statically configurable at compile/link time, to support the linking of Angel into the application. The approach to all aspects of configurability in Angel is that the configuration choices have to be made at compile/link time, not at runtime. For example, the memory map, exceptions handlers and interrupt priorities are all fixed at compile/link time and cannot be reconfigured at runtime.

#### Exception vectors

Exception vectors are initialized statically by Angel, and are not written to after this. This is necessary to support systems with ROM and Address 0, where the vectors cannot be overwritten.

### 6.4.2 Communications

Angel communicates using ADP, not RDP, and makes use of channels to allow multiple independent sets of messages to share a single comms link. Angel supports an error-correcting communications protocol over the link.

The following links are supported:

- serial and serial/parallel connection from host to PIE or PID board, with Angel resident on board in ROM
- Ethernet connection from host to PID board, with Angel resident on board in ROM (this requires Ethernet enhanced Angel)
- host connection to EmbeddedICE box (serial and/or parallel) using ADP, EmbeddedICE JTAG connection to EmbeddedICE Macrocell, no Angel present
- (ARMTDMI only): Host connection to interface box (serial and/or parallel) using ADP, EmbeddedICE JTAG connection to EmbeddedICE Macrocell which communicates with Angel resident on the target board over the Debug Comms Channel, using ADP. Note that this requires different software on the EmbeddedICE board. This software is supplied.

### 6.4.3 Channels

Angel supports flow control over the link. The Device Driver Architecture enables multiple packets to be processed simultaneously, using a serialization system, and does not disable interrupts for long durations. Therefore, overflow should not be an issue, under normal circumstances.

Angel also supports multiplexed communications channels for your programs over the debug communication link. For example, this allows the operating system to communicate with the debugger.

Angel device drivers support an interface that allows multiple channels to be opened onto a single device. It then manages these transparently to higher-level software, such as Angel itself, or your application.

An application need not know that a device is used by other systems using other channels. Thus, late in the development cycle, you can remove the multiplexed channels support from the final device drivers, leaving a minimal single channel device driver without having to modify the application.

### 6.4.4 Reliable channels and buffer management

Components of the host system communicate with components of the target system using multiple independent *channels* along the single physical link joining host and target. The host and target system channel managers make sure that this multiplexing of different logical channels operates reliably.

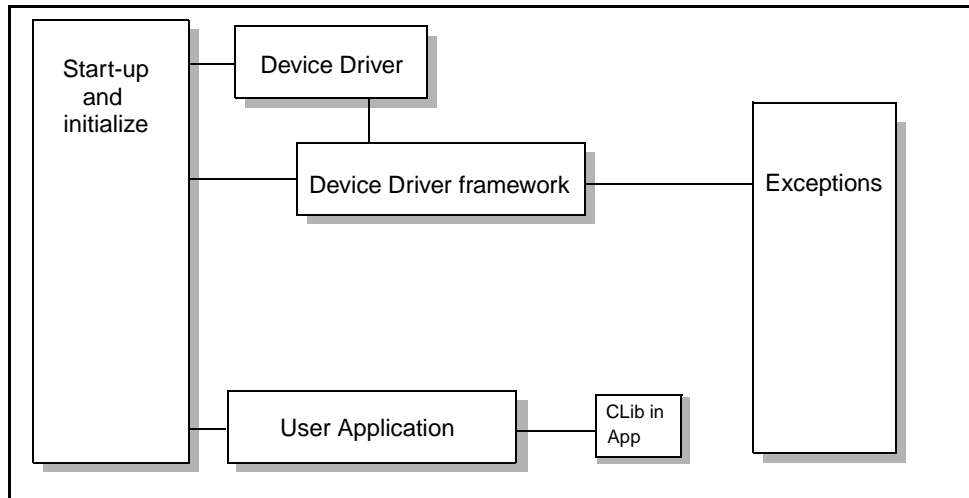
Device drivers in the host and target systems transmit and receive individual data packets over the physical link. Although the device drivers detect and reject any corrupted data packets, it is left to the channel managers to monitor the overall flow of data.

The channel managers store transmitted data in buffers, in case retransmission is required.

## 6.4.5 Device drivers

Angel allows you to write drivers so that you can use alternative devices for debug communication (for example, a ROMulator), and is structured to allow easy extension to support different peripherals. Also, an application can address devices directly.

The Device Driver Framework does not support channels—it just despatches interrupts:



**Figure 6-3: Device Driver Framework**

## 6.4.6 Applications

Angel supports the debugging of real-time applications, and is designed to minimize the time for which interrupts are disabled.

Angel minimizes interrupt latency by ensuring that as little processing as possible is carried out with interrupts disabled. After the first byte of a packet has arrived, interrupts must no longer be kept disabled. Instead, it is necessary to switch modes, set up a suitable stack, enable interrupts, and *only then* poll for the rest of the packet. If interrupts arrive for other devices during this time, they are serviced straight away.

The debug protocol interpreter can be used by other applications and operating systems.

### Applications in ROM: Late Debugger Startup

Angel supports applications in ROM, where the application has the capability to call the debugger and request assistance, rather than the debugger assuming control at initialization. This is known as Late Debugger Startup.

ROM applications can run from anywhere in ROM, although the address must be specified at link time. Angel can be reconfigured for ROM at 0 and other systems.

## 6.4.7 Semihosting

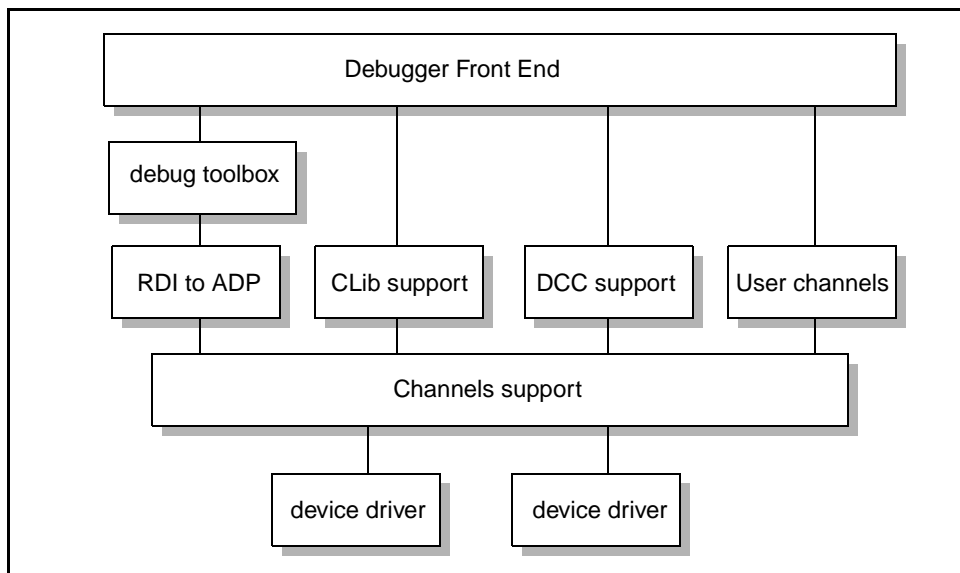
Semihosting need not be used and can be turned off if desired.

## 6.5 Debuggers

Angel is supported by the following host debuggers:

- ARM Debugger for Windows (NT, 95)
- armsd (Unix, NT, 95)

**Figure 6-4: Debuggers** shows the structure of the debuggers (ignoring RDP compatibility):



**Figure 6-4: Debuggers**



## 6.6 Minimal Angel

The minimal Angel library is intended to support the later stages of debugging. It does not support debugging via ADP, semihosted C library support, and so on. However, it does supply everything needed by an application which has been developed to use the `devappl.h` API to access raw devices, and it does supply a startup and interrupt framework that is compatible with full Angel.

### 6.6.1 Components of minimal Angel

The minimal Angel library contains almost the same initialization code, interrupt handling and exception handling as a full version of Angel. The device driver architecture is the same, and any Angel device driver that can be compiled as a 'raw' device is fully supported.

The minimal library contains sufficient components to allow it to be swapped for a full library. The main difference (unless late debugger startup has already been used) is that an image containing an application and the minimal library will initialize and then immediately enter the application (at `__entry`).

The minimal library does *not* contain a serializer, ADP debugging support, semihosting support, an Undefined exception handler, channels support, or reliability and retransmission support.

The minimal library is approximately three to five times smaller than the full library. The actual size depends on which device drivers are included and on compile-time debugging and assertion options.

### 6.6.2 Building a minimal Angel library

Because there are major differences between object files for a full library and a minimal library, separate build directories and makefiles are provided for minimal Angel.

For example, the directory and makefile under UNIX (SunOS) with Gnu C, for building full Angel targeted to a PIE board, are:

```
angel/pie.b/gccsunos/  
angel/pie.b/gccsunos/Makefile
```

and for minimal Angel:

```
angel/pie.min/gccsunos/  
angel/pie.min/gccsunos/Makefile
```

Within the Angel source code, differences between the full and minimal cases are controlled by the `MINIMAL_ANGEL` macro, which is set to 0 for full Angel and to 1 for minimal Angel.

## 6.7 Porting Angel to New Hardware

Angel has been designed to make porting to new hardware as easy as possible. However, there are many configurable features of Angel, and code, that inevitably require modification.

When you start to port Angel to another board, select the version of Angel for a board most similar to your board. The most significant features in this decision are likely to be:

- device drivers

If possible choose a version which uses the same or very similar communications hardware. This simplifies modifying the device driver code.

- Cache/MMU

The PID and PIE ports do not use a cache or MMU. The cogent and ebsa110 ports do.

Once you have selected a port of Angel to base your own port on, such as PIE, copy the board-specific directories. For example, for the PIE port you would need to copy `pie.b` and `pie`.

### 6.7.1 Target-specific files

You should look through the following files and modify them for your system:

<code>devconf.h</code>	<p>This is the main configuration file. It includes device declarations, memory layout details, stack sizes, and interrupt source information (IRQ versus FIQ).</p> <p>You should check every item in this file and make sure it is set up correctly for your board.</p>
<code>devices.c</code>	<p>This includes the headers for device drivers for the system, and so needs to be modified if you add, remove, or rename any device drivers.</p>
<code>banner.h</code>	<p>This declares what board Angel is running on, and with what options.</p>
<code>target.s</code>	<p>This provides important startup macros specific to the hardware. You should look at each macro in this file and if necessary change them for your board.</p> <p>It also includes the <code>GETSOURCE</code> macro, which is used to identify which interrupt source has caused an interrupt. This needs to be modified to suit the interrupt-driven devices and interrupt scheme used on your board.</p>
<code>makelo.c</code>	<p>This is built into <code>makelo</code>, which reads various <code>.h</code> files and produces a <code>.s</code> file. This allows assembler and C modules to access global constants without having to have two copies (one for assembler use and one for C use).</p> <p>If you introduce new constants that need to be shared by C and assembler, make sure you add them to this file.</p>

All other files in the target specific source directory are device driver sources. You may need to modify these even if your board uses the same communications chips as the port you are basing your port on. If you are using different communications hardware, you should go through these files thoroughly and recode for your own hardware.

## 6.7.2 Modifying the Makefile

You must modify the Makefile so that it uses your directories, compiles and assembles your source, and links your object files. In addition, there are various major configuration options set in the Makefile such as:

THUMB_SUPPORT	Defines whether or not the target ARM has Thumb Support.
MINIMAL_ANGEL	Minimal Angel is a system which does not support debugging. This option should always be left set to 0, because a separate Makefile and build area is provided for building Minimal Angel systems.
WADDR	Defines where Angel should store its read/write data.
ROADDR	Defines where Angel should run from. If this is set to the address of the ROM, Angel executes from ROM. If it is set to a RAM address, Angel can still be put into ROM, but copies itself into RAM and executes from there (this is useful when ROM is very much slower than RAM).

EmbeddedICE is an invaluable tool for helping you to debug Angel, because it can operate even before the basic Angel functionality is working, for example, when device drivers are not yet working. However, not all ARM processors allow use of EmbeddedICE (the `DI` debug extensions are needed), so other options are ROM Emulators and Logic Analysers. Included with Angel are files to help you use an E5 ROM Emulator (Crash Barrier Ltd) or a neXus ROMulator (neXus Ltd).

## 6.7.3 Configuring where Angel runs

This section briefly describes how to configure Angel to run from ROM, ROM mapped at zero, or RAM.

### Link addresses

The Makefile for `angel.rom` contains two Makefile macros that control the addresses where Angel is linked:

RWADDR	This defines the base address for read/write AREAs, ie. <code>dataseg</code> and <code>bss</code> (zero-initialized) AREAs, along with some assembler AREAs.
ROADDR	This defines the base address for read-only AREAs, ie. <code>code</code> AREAs.

The target-specific configuration file `devconf.h` contains a number of macros which define the memory layout of the target board, and checks to make sure that `RWADDR` and `ROADDR` contain sensible values. Most of these macros are only used within `devconf.h`.

(for the sanity checks, in the `READ/WRITE_PERMITTED` macros, and for defining application stack and heap areas), but the macro `ROMBase` is also used during startup to calculate the offset between the code currently executing in ROM and its eventual `ROADDR` destination.

## ROM locations

Angel can support two types of ROM system:

- ROM is mapped to address 0 upon reset, and mapped out during Angel bootstrap
- ROM is permanently mapped to address 0

For the first type:

- 1 Define `ROMBase` (`devconf.h`) as the 'normal' (ie. mapped-out) address of the ROM.
- 2 Set the ROM-only build variable (`target.s`) to `FALSE`.
- 3 Provide an assembler macro called `UNMAPROM` in `target.s` which maps the ROM away from 0.
- 4 Declare the Makefile macro `FIRST` as `'startrom.o(ROMStartup)'`, including the quote (') characters.

For the second type:

- 1 Define `ROMBase` (`devconf.h`) as 0.
- 2 Set the ROM-only build variable (`target.s`) to `TRUE`.
- 3 Declare the Makefile macro `FIRST` as `'except.o(__Vectors)'`, including the quote (') characters.

## Processor vectors

Regardless of where you declare `RWADDR` and `ROADDR` to be, the ARM processor requires the vector table to be located at zero. There are a number of situations in which this happens by default, for example when `RWADDR` is set to 0, or in ROM-at-zero systems.

When this does not happen by default, Angel explicitly copies everything in `AREA __Vectors` from `RWADDR` to zero. All code within this `AREA` should be position-independent, because it is linked to run at `R[OW]ADDR`, not zero.

In most configurations, Angel is able to detect a branch through zero by application code, and report it as an error. However, this is not possible in ROM-at-zero systems where a branch through zero would cause a system reboot (or, more likely, a system crash, because the reboot code expects to be executing in a privileged mode).

## 6.8 Downloading a Debug Agent

Angel and EmbeddedICE can both download a new version of themselves. You can do this from the command line in the ARM debuggers, using the `loadagent` command, or in the ARM Debugger for Windows, using a menu selection.

Downloading a new debug agent is often preferable to replacing the ROM because it is usually quicker, and does not involve physically dismantling an EmbeddedICE box or removing a ROM from its socket and reprogramming it using an EPROM programmer. However, downloading a new agent is not a permanent measure—if the board is powered down, or even just reset, the downloaded debug agent is lost.

### EmbeddedICE

EmbeddedICE always has enough free memory to allow a new debug agent to be downloaded. EmbeddedICE can download two different debug agents:

- A new copy of the EmbeddedICE software (see the chapter **Chapter 7, EmbeddedICE** for details and restrictions).
- The ADP over JTAG software for the EmbeddedICE, which acts only as a ‘go between’, passing messages from the host debugger to Angel running on the target. As a result, any further `loadagent` commands are directed to Angel rather than to the EmbeddedICE box. To revert to standard EmbeddedICE operation, reset the EmbeddedICE box. See also **6.12.2 ADP over JTAG using EmbeddedICE board** on page 6-28.

### Angel

Unlike EmbeddedICE, Angel is not always capable of downloading a new copy of Angel and then restarting. To do this, Angel needs enough spare RAM to copy the entire new Angel into RAM before relocating it and running it.

Also, Angel is built to relocate a downloaded new Angel to the address that the new Angel was built to execute from, and then to execute it. (See **6.7.3 Configuring where Angel runs** on page 6-19.)

So, for example, an attempt to download a copy of Angel which expects to run from ROM space would fail, because Angel attempts to copy this image into ROM (which fails) and then execute it.

## 6.9 Adding a Channel

In the ARM Software Development Toolkit version 2.1, it is not possible to create a new Angel channel. However, it is possible to re-use two of the unused channels. The following instructions use the Debug Comms Channel (TTDCC) in the Angel source code as an example. You should adapt this suitably for your own channel.

- 1 Releases after 2.1 may allow the addition of extra channels. If you are adding a new channel, rather than re-using an unused one, add the channel to `enum ChannelIDs` in `angel/chandefs.h`. For example, `CI_TTDCC` is already defined.
- 2 If the channel is to be used with ADP-style messages, add suitable entries in your own file, based on `angel/adp.h`. Use the `ADPREASON()` macro to encode the channel into the reason code. For example, `ADP_TDCC_ToHost` and `ADP_TDCC_FromHost`.
- 3 `CI_TTDCC` is a target-controlled channel. The target side is implemented in `iceman2/iceb.c`, in functions `do_cc_transaction()`, `CC_ToHost()` and `CC_FromHost()`. For blocking transactions, you should:
  - a) Build a message, probably with `msgbuild()`. If the message follows ADP conventions, it should start with a reason code, with the TtoH bit set.
  - b) Send it and wait for a reply with `angel_ChannelSendThenRead()`, specifying the device (which will usually be `CH_DEFAULT_DEV` to indicate the default) and the channel (`CI_TTDCC` in this example).
- 4 On the host side, a handler must be registered for the new channel. This is done with `Adp_ChannelRegisterRead()`. See `angsd/ardi.c`. The `angel_check_DCC_handler()` function registers the handler for `CI_TTDCC`, and `HandleDCCMessage()` is the handler.
- 5 Whenever a message arrives for the channel, it is passed to the registered handler. Within the handler, unpack the message (`unpack_message()`), and decode its reason code. Take appropriate action and then send a reply on the same channel, with the same reason code, but with the HtoT bit set. The function `msgsend()` does it all, or you can use `msgbuild()` and `Adp_ChannelWrite()` if you require more flexibility in assembling the message.
- 6 If nonblocking behaviour is required at the target, or if the channel is host-originated, the steps needed on the target are very similar to steps 4 and 5, except that the call to register a handler is `angel_ChannelReadAsync()`.

## 6.10 Using Angel on the ARM PID7T board

### 6.10.1 Links

There are a number of links on the ARM7T PID board which must be set up appropriately for use with Angel:

Link 18: Remap ROM link

In: ROM at 0, can be remapped  
Out: ROM at high address only

Make sure you set this link In for use with Angel running from ROM or Flash, and Out if EmbeddedICE is being used to download Angel into RAM before running it. This link determines whether ROM appears at 0 on reset.

Link 3 (5-6 2nd link nearest to the headercard in block of 4): EPROM Select link

In: Flash selected (device is writable)  
Out: EPROM selected (device is read-only)

This should be set correctly, according to whether you are using Flash or EPROM. This is particularly important if you intend to use the Flash download utility (see **6.10.4 Flash downloader** on page 6-24).

Link 3 (7-8 nearest link to the headercard in block of 4) 8/16 bit select

In: 16 bit EPROM/Flash selected  
Out: 8 bit EPROM/Flash selected

This should be set correctly for the type of EPROM/Flash device. U13 holds a 16-bit device and U12 holds an 8-bit device.

### 6.10.2 Parallel port adaptor

The ARM7T PID board requires an adaptor card to connect to a standard parallel cable. Although a standard parallel cable connects to the ARM7T PID board as supplied, parallel communications do not work. The adaptor board plugs into the PID7T parallel port, and requires a ribbon cable connection between the adaptor board and connector POD6 on the PID7T board. If you do not have an adaptor board and you need to use the parallel port, refer to the documentation you received with your PID7T board, and if necessary contact your supplier to obtain an adaptor card.

### 6.10.3 Ethernet

The Fusion stack has been compiled to support five sockets, and the maximum size of IA and UDP packets is 2048 bytes. These limits result in a 22KB Fusion heap. These limits cannot be changed by users of Angel unless a source licence for Fusion is obtained.

## 6.10.4 Flash downloader

The Flash downloader is a utility provided with the ARM Software Development Toolkit, and integrated into the ARM Debugger for Windows. You can use it to program a Flash RAM while it is on the board. This works only if Angel is running from RAM (the default) at the time, or if EmbeddedICE is being used rather than Angel. The correct version for the endianness of the board should be used. (See **6.10.6 Big- and little-endian operation** on page 6-24.)

The Flash downloader attempts to recognise the Flash RAM being used, and will fail if it does not recognise it, because it must understand how to program it. If you are using armsd, the flash downloader should be passed either the argument `-e` or the name of the file to be downloaded into flash.

If the file is being downloaded, you are prompted for the sector from where the programming should start. Angel should be programmed into the start of the Flash RAM, ie. from sector 0.

Alternatively, the Flash downloader program can be used to override the default IP address and net mask used by Angel for Ethernet communication. To do this from armsd, pass the Flash download program the argument `-e`. The program prompts for the IP address and net mask. If you are using the ARM Debugger for Windows, select the appropriate option from the menu.

## 6.10.5 Using an EPROM programmer

If you use the Flash download program you do not need to worry about how the code is stored in memory. However, if you use an EPROM programmer to program the device, this may be important. In general, programming little-endian code into 8 and 16 bit devices, or big-endian code into 8-bit devices does not cause any problems. If you need to program big-endian code into 16-bit devices using an EPROM programmer you may need to swap even and odd bytes to get the device to program as required. Most EPROM programmers allow you to do this.

To check, use EmbeddedICE to examine the memory and verify that instructions are being read correctly.

## 6.10.6 Big- and little-endian operation

If you are running a little-endian version of Angel and wish to change to a big-endian version, you can do this by running the Flash download program. However, because of an incompatibility between the way big- and little-endian code is stored in 16-bit wide devices, this works only if the target device is an 8-bit Flash device.

- 1 Make sure you are using the 8-bit Flash device (U12).
- 2 Start little-endian Angel by switching on the board and connecting to the debugger.
- 3 Run the Flash download program and program the Flash with the big-endian Angel image. This works because Angel operates out of SRAM.
- 4 Quit the debugger and switch off the board.
- 5 Change the EPROM controller (U10) to be the big-endian controller.



- 6 Insert the BIGEND link (LK4).
- 7 Power up the board and connect the debugger. Make sure that the debugger is configured for big-endian operation.

Once you have a big-endian Angel in Flash, you can use a big-endian version of the Flash downloader to program a new copy of Angel into the 16-bit device. To do this:

- 1 Switch on the board.
- 2 Start the debugger.
- 3 Insert the SEL8BIT link (LK6-4) so that the target device is now the 16-bit Flash chip.

Remember that you must provide a 16-bit wide Flash device, because one is not supplied with the board.

**Note** *There is no performance gain to using a 16-bit wide device in this case, because Angel copies itself to SRAM and executes from there.*

## 6.11 Using Angel on the ARM60 PIE Board or ARM7 PIE Board

### 6.11.1 Interrupts

The ARM60 PIE board as supplied has serial interrupts wired to FIQ. Angel for this board works with serial interrupts on either IRQ or FIQ. For your application, you may want to change serial interrupts to occur on IRQ, in which case you need to resolder link JP1. Refer to the documentation for the ARM60 board.

The ARM7PIE board comes with serial interrupts wired to IRQ, and Angel works with this setup.

### 6.11.2 Big- and little-endian operation

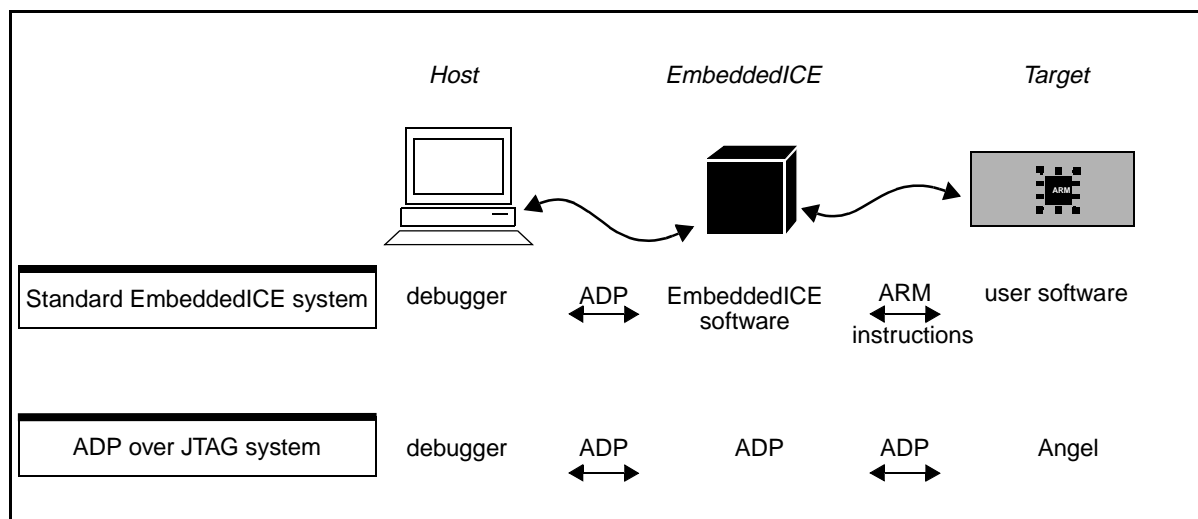
Both the ARM60 and ARM7 PIE boards are supplied in little-endian configuration. To change to big-endian configuration:

- 1 Rebuild Angel for big-endian operation
- 2 Use the `wordswap` utility (source for which is provided with Angel). This switches the byte order of your ROM image so that it is suitable for blowing into an EPROM for use on these two boards.
- 3 Change links:
  - on the ARM60 PIE board, change the position of link JPS to indicate big-endian operation
  - on the ARM7 PIE board, resolder link 2 (refer to the ARM7 PIE board documentation)

## 6.12 The Debug Comms Channel, Angel, and EmbeddedICE

The Debug Comms Channel can be used in two distinct ways:

- an additional channel for EmbeddedICE (no Angel)
- Angel with communication over the Debug Comms Channel (ADP over JTAG)



**Figure 6-5: Debug Comms Channel, Angel, and EmbeddedICE**

### 6.12.1 An additional channel for EmbeddedICE (no Angel)

The ADP-compatible EmbeddedICE has two channels that deal with data on the Debug Comms Channel. This makes it cleaner to process at both ends. It is also considerably easier to use the Debug Comms Channel from the host end, because you can replace the standard handler which just reads and writes data to or from a file.

On the target, you must provide code to drive the Debug Comms Channel directly using coprocessor instructions. Angel is not required to be running on the target.

On a UNIX system under `armsd`, the `ccin` and `ccout` commands can be used to direct output from the Debug Comms Channel to a file or take input for the Debug Comms Channel from a file.

If you are using the ARM Debugger for Windows, a channel viewer window can be opened that allows direct input and output via the Debug Comms Channel (see **3.3.3 Remote\_A** on page 3-4 for more information).

## 6.12.2 ADP over JTAG using EmbeddedICE board

In a standard EmbeddedICE system, the EmbeddedICE software acts as the Debug Agent that processes debug requests issued by the host. EmbeddedICE interacts with the target processor by halting execution (entering debug state) and then forcing the ARM processor to run instructions which read and write the information the debugger requires. As a result, the ARM processor is halted for significant periods of time, which can cause problems for real-time applications.

Alternatively, you can use the Debug Comms Channel as a communications link. This allows the debugging of real-time systems which have no other comms links. In this system, called 'ADP over JTAG using EmbeddedICE', Angel is required on the target, and uses the Debug Comms Channel in the same way as it would a serial link (see **Figure 6-5: Debug Comms Channel, Angel, and EmbeddedICE**). In this system the processor is never put into debug state and so is likely to be more suitable for real-time applications.

To use this 'ADP over JTAG using EmbeddedICE' system, a version of Angel that supports the Debug Comms Channel is needed on the target system, and a separate ROM image for EmbeddedICE is also required (this is the ADP over JTAG software).

The ADP over JTAG software can be put into ROM on EmbeddedICE or downloaded onto EmbeddedICE, using the `loadagent` command in `armsd` or a configuration option in the ARM Debugger for Windows. The ARM debuggers can be used in exactly the same way as they are used when connected to a target running Angel over a serial link.

For more information about downloading a new debug agent, see **6.8 Downloading a Debug Agent** on page 6-21.

## 6.13 Notes for Demon Users

This section outlines the differences between Angel and Demon, Angel's predecessor.

### **Demon uses RDP whereas Angel uses ADP**

RDP is the debug protocol used by Demon. ADP is the new debug protocol used by Angel. The two protocols are incompatible. You must invoke armsd differently, and select Angel in the ARM Debugger for Windows.

At one level, ADP has changed little from RDP—there are still requests to read and write memory and register, set breakpoints, and so on. However, the new protocol is the bottom layer of a stack of protocols, the whole of which is referred to as ADP. Above the debug message layer is a channel distribution layer, and above this a device-dependent transmission layer—a thin and simple layer when using serial communications, but much more complex when using Ethernet-based communications.

### **Angel uses a new, incompatible semihosting SWI scheme**

A different ARM C library must be used with Angel. To make a semihosted program that works with Demon work on Angel, you must relink the program with the Angel C library, not the Demon C library.

In the ARM Software Development Toolkit version 2.1, the Angel C library is the default, whereas earlier releases used the Demon C library.

### **Angel uses channels**

In an Angel system there are channels: one for host-originated debug messages, one for target-originated debug messages, one for semihosting support, and so on.

As a result, semihosting is not part of the debug protocol, but is encoded in a separate (but similarly constructed) protocol used on a different channel.

Unlike RDP, ADP does not have states in which the protocol can 'get stuck', or messages cross unexpectedly. This is mainly due to the channel system used for ADP.

### **All messages get an immediate response**

For example, Execute is responded to with Execute acknowledge. This was not true for RDP.

### **ADP is packet-based**

Demon's RDP was a pure bytestream system. Angel's ADP can cope with missing or corrupt packets.

### **ADP messages have thread identifiers**

This potentially allows ADP to be used in multi-threaded systems (although this is not used yet). RDP does not have this capability.

## **Angel is implemented in C wherever possible**

The Angel source code is much better structured than the Demon source code, and is mostly written in ANSI C. Only a few routines are written in Assembler, mainly because it is simpler to use Assembler for functions such as exception handlers.

Advanced users of Angel can replace modules with their own code, provided that it conforms to the interfaces specified for that module.

## **Angel has a device driver architecture and mini task-scheduler**

Angel includes an extensible device driver scheme that enables advanced users to add their own device drivers to Angel much more easily than with Demon.

To process device-related interrupts, debugging requests, application tasks and callbacks, a mini task-scheduler is included in Angel. This does not allow multi-threaded applications to run under Angel, but makes it possible for Angel to process a number of requests arriving very closely together such that it does not have time to complete them all before the next one arrives.

## **Angel is easily portable to different boards**

It is easy to change board-specific details such as the memory map that Angel assumes the board is using. (This was not easy with Demon.)

## **Angel is easier to use with third-party operating systems**

Angel separates out various types of operation (such as accessing application CPU and memory state, setting breakpoints, exception handling) which may need to be reimplemented if a third party operating system is in use.

## **Angel is easier to cut down for use on production hardware**

Minimal Angel makes it easier to move from development hardware to production hardware. This is because Angel can be recompiled to eliminate most of the functionality (eg. debug support, semihosting support and so on) that is no longer needed.

## **Angel does not work with 26-bit configured ARMs**

Unlike Demon, Angel does not work with 26-bit configured ARMs. This allows a number of complexities with 26-bit exceptions to be ignored, reducing the size and complexity of Angel as a whole.

## 6.14 Example of an Application Device using Angel (UNIX only)

Angel provides a framework to the application that permits the application both to use dedicated (raw) devices not being used for debugger communications, and to share devices being used by Angel to communicate with the host.

This section describes two example programs:

- the `appldev` program is a simple example of a target application using devices
- `tstmain` is a UNIX program designed to talk to `appldev`

Locations and descriptions of the source code for this example are given in **6.14.4 Source code layout** on page 6-37.

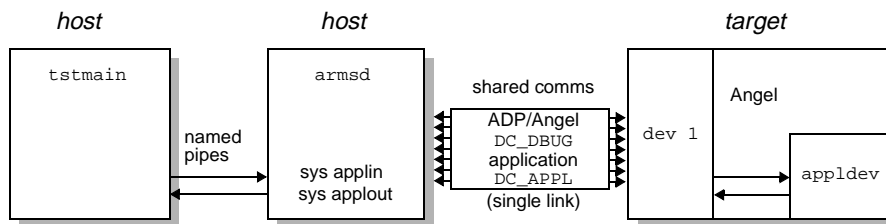
### 6.14.1 Modes of operation

There are three main configurations to consider:

- full Angel, shared device
- full Angel, separate raw device
- minimal Angel

#### Full Angel, shared device

This is the situation during early development on a target with a single communications device, which at this stage must be shared between Angel and the application:



On the target, `appldev` can either be downloaded via Angel, or linked in with Angel as a ROM.

In some ways this is the most complicated case, but it can be simplified if you visualize `armsd` and Angel working together to create a virtual connection between `tstmain` and `appldev`. `tstmain` communicates over the named pipes as if they connected directly to `appldev`.

At the target, Angel takes care of multiplexing its own communications with the application's communications, and presents a standard API to the application.

# Angel

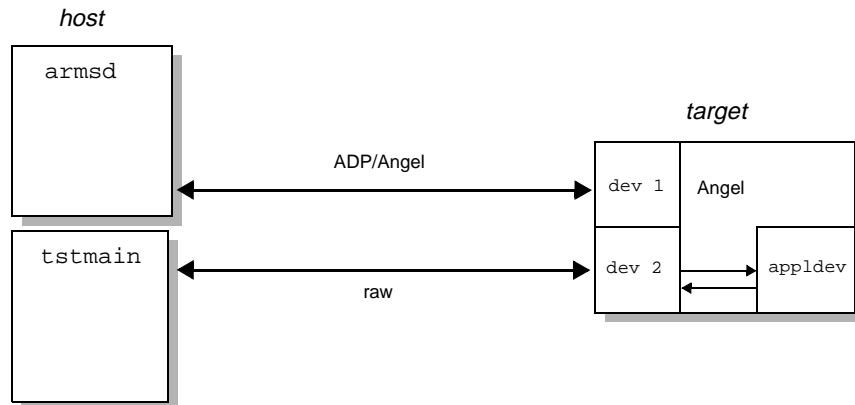
On the host, `armsd` intercepts the multiplexed application communications. The `sys applin` and `sys applout` commands allow input to the application and output from the application to be directed from and to files (named pipes in this example).

`tstmain` runs as a separate process on the host, and communicates via the named pipes connected to `armsd`. The `tstmain` program is unaware that it is not communicating with `appldev` over a physical device.

See **Full Angel, shared device** on page 6-33 for information about how to set up this example system.

## Full Angel, separate raw device

When more than one device is available, one can be dedicated to Angel and another can be used by the application. For example, you could use a PID board with two serial ports, or Ethernet and serial port, or DCC and serial port, and so on.



On the target, `appldev` can either be downloaded via Angel, or linked in with Angel as a ROM. `armsd` and `tstmain` need not be running on the same host machine.

Device 1 is used by Angel for ADP communications with `armsd`. Device 2 is used by the application in raw mode: no packeting or error correction is provided. Angel simply provides a thin veneer between the application and the device driver.

From the application's point of view, the API and the communications that take place are just the same in all cases.

A further possibility (not illustrated here or by `appldev`) is that the application handles the second device entirely independently of Angel, so that the device driver is part of the application itself.



## Minimal Angel

Later in development, when you do not require Angel debugging, the application can be linked with a minimal Angel library which provides the same device interface as the full Angel. As long as the Angel API and drivers offer sufficient performance and flexibility, there is no need to recode the application's communications, or to write new device drivers.



The minimal Angel simply provides a framework for the device drivers, and a thin veneer between the device driver and the application. It also handles initialization (stack, and so on) at startup.

### 6.14.2 Running appldev and tstmain

For this release, the host side of this example has been implemented only on UNIX. It assumes that two terminal windows are available, one for `armsd` and one for `tstmain`. The following notation is used:

A> shell prompt in first window  
 B> shell prompt in second window  
 armsd: armsd prompt in second window

See **6.14.4 Source code layout** on page 6-37 for details of the directory structure holding the example sources and executables.

#### Full Angel, shared device

This example can be used with the distributed version of Angel, because it requires only that the device being used for Debugger/Angel communication can also be shared by the application, and this is the case for all Angel devices.

- 1 Create two named pipes:
 

```

      A> /etc/mknod rxpipe p
      A> /etc/mknod txpipe p
      
```
- 2 In one window, start up `tstmain`, specifying the two pipes:
 

```

      A> tstmain rxpipe txpipe
      
```
- 3 Reset the target system (or otherwise make sure Angel is running and in control).
- 4 In the second window, make sure that the `appldev` example program has been

built. To do this, make the current directory the build directory for the board you are using and the host you are using: for example: `angel/pid.b/gccsunos`. Then:

```
B> make appldev
```

- 5 Start up `armsd`, specifying the device and options as appropriate:

```
B> armsd -adp -port <device> <options...>
```

For example:

```
B> armsd -adp -port 1 -line 38400
```

- 6 If you are using `appldev` as an application, download it:

```
armsd: load appldev
```

- 7 Open the connections between the named pipes and the application via the shared device. The order of these two commands is important:

```
armsd: sys applout rxpipe
```

```
armsd: sys applin txpipe
```

- 8 Start the application:

```
armsd: go
```

- 9 Observe output from `tstmain` in the `tstmain` window and from `appldev` in the `armsd` window.

- 10 Interrupt the application after the first five cycles or so:

```
<Ctrl-C> (to armsd)
```

- 11 Either quit from `armsd`, or close the connections:

```
armsd: sys applin
```

```
armsd: sys applout
```

In either case, `tstmain` will exit because the named pipes have been closed.

- 12 You may have to remove and recreate the named pipes before you re-run.

## Full Angel, separate raw device

This example can be run only when Angel has a device configured to be raw. In other words, it can be run when a device is not configured for Angel-to-debugger communication, only for applications using the Angel raw device interface. By default, no devices are built in this way, so you must rebuild Angel. The following steps take you through rebuilding Angel for the PID7T board in order to use Serial B as a raw device.

- 1 Edit `pid/devconf.h`, and change the line defining `ST16C552_NUM_PORTS` so that two serial ports are supported. By default the second port will be a raw device.

- 2 To rebuild Angel, remove all object files and derived files. For example:

```
B> rm *.o ../lolevel.s makelo appldev.*
```

- 3 Now invoke the Makefile:

```
B> make
```

- 4 Rebuild the `appldev` example:

```
B> make appldev
```

- 5 Start up `armsd`, specifying the device connecting to Angel, and appropriate options:  

```
B> armsd -adp -port <device> <options...>
```

For example:

```
B> armsd -adp -port /dev/ttya -line 38400
```
- 6 Load and run the new version of Angel that you have just built:  

```
armsd: loadagent angel.rom
```
- 7 Download the newly built `appldev` example program:  

```
armsd: load appldev
```
- 8 In the other window, start up `tstmain`, specifying (twice) the device connecting to the application:  

```
A> tstmain <device> <device>
```

For example:

```
A> tstmain /dev/ttyb /dev/ttyb
```
- 9 Reset the target system (or otherwise ensure Angel is running and in control).  
In the first window, start the application running under Angel:  

```
armsd: go
```
- 10 Observe output from `tstmain` in the `tstmain` window and from `appldev` in the `armsd` window.
- 11 Interrupt the application after the first five cycles or so:  

```
<Ctrl-C> (to armsd)
```

## Minimal Angel

- 1 Start up `tstmain`, specifying (twice) the device connecting to the application:  

```
A> tstmain device device
```

For example:

```
A> tstmain /dev/ttya /dev/ttya
```
- 2 Start the application image by running `angel/pid.min/appldev`. Do this either by putting the image into ROM or Flash, or by downloading it using EmbeddedICE. If you are using EmbeddedICE, load the image to the address `ROADDR` as defined in the Makefile. In this case also be sure to switch off `$semihosting_enabled` and `$vector_catch`.
- 3 Observe output from `tstmain`.
- 4 Interrupt `tstmain` with `<Ctrl-C>` when you have finished.

## 6.14.3 Notes about the example code

`appldev` is intended to represent an embedded application, polling a connected host for commands relating to the internal configuration of the application. When polled, the host can return an idle command or one of:

<code>RESET_DEFAULT_CONFIG</code>	the target should reset to its defaults
<code>GET_CONFIG</code>	the target should send its current configuration to the host, which will reply with <code>GET_CONFIG_ACK</code> to finish the transaction
<code>SET_CONFIG config</code>	the target should use the supplied configuration


For the purposes of the example, a configuration consists simply of a variable-length string.

`tstappl.c` implements the host-side functionality. `test_dc_appl_handler()` is called to process input from the target. If the input is a poll request, `process_poll()` is used to decide what to do next.

`process_poll()` simply keeps count of the number of polls and returns various commands for the first five polls:

- 1 Get current configuration (should be default).
- 2 Set a new configuration.
- 3 Get current configuration again (should be new).
- 4 Reset default configuration.
- 5 Get current configuration again (should be default).

`tst_main.c` provides UNIX command-line processing and device interfaces. If the devices are serial ports, appropriate setup is performed to match Angel defaults (9600 baud, 8 data bits, 1 stop bit, no parity). Once `tst_main.c` has opened the input and output devices, it sits in a loop passing input to `test_dc_appl_handler()`.



# User Guide

ARM DUI 0040C



# User Guide

ARM DUI 0040C

`angel/target.min/hostos/angel.lib`  
minimal Angel library

`angel/target.min/hostos/appldev`  
minimal Angel and `appldev` linked as a ROM image. You need to 'make' this.

`angsd/`  
sources for Angel-specific parts of host debugger needed by `tstmain` example program

`angsd/examples/`  
host example sources

`angsd/examples/tstappl.c`  
core of `tstmain`, which communicates with `appldev`

`angsd/examples/tstappl.h`  
header for `tstappl.c`

`angsd/examples/unix/`  
UNIX-specific example sources

`angsd/examples/unix/tstmain.c`  
UNIX framework for `tstappl.c`

`angsd/examples.b/hostos/`  
host example build directory eg. `angsd/examples.b/gccsunos`

`angsd/examples.b/hostos/Makefile`  
Makefile for `tstmain`

`angsd/examples.b/hostos/tstmain`  
UNIX executable. You need to 'make' this.

# 7

## EmbeddedICE

7.1	What is EmbeddedICE?	7-2
7.2	The Effect of EmbeddedICE on the Debuggee	7-5
7.3	Connecting and Powering Up	7-6
7.4	Configuring EmbeddedICE to Match your Target Core	7-7
7.5	Configuring the Debugger	7-9
7.6	Watchpoints and Breakpoints	7-13
7.7	Vector Breakpoints and Exceptions	7-15
7.8	Semihosting	7-17
7.9	Reset and JTAG Signal Connection	7-20
7.10	Debugging Applications in ROM	7-24
7.11	Accessing the EmbeddedICE Macrocell Directly	7-28
7.12	EmbeddedICE and Target Board Memory Layout	7-30
7.13	Timer Accuracy	7-32
7.14	Floating-Point and Other Coprocessors	7-39

# EmbeddedICE

---

## 7.1 What is EmbeddedICE?

EmbeddedICE is an extension to the architecture of the ARM family of RISC processors, and provides the ability to debug cores that have been deeply embedded into systems. It consists of three parts:

- a set of debug extensions to the ARM core
- the *EmbeddedICE macrocell*, to provide access to the extensions from the outside world
- the EmbeddedICE interface to provide communication between the EmbeddedICE macrocell and the host computer

### 7.1.1 Debug extensions to the ARM core

The presence of debug extensions to the ARM core are indicated by the **D** suffix on the core name. The extensions consist of a number of scan chains around the core and some additional pins that are used to control the behavior of the core for debug purposes.

The three most significant pins are:

- |         |   |
|---------|---|
| BREAKPT | allows external hardware to halt execution of the processor for debug purposes. When HIGH, the current memory access is tagged as breakpointed. If the memory access is an instruction fetch, the core enters the debug state if and when the instruction reaches the execute stage of the pipeline. If the memory access is for data, the core enters the debug state when the current instruction completes execution. The EmbeddedICE macrocell has control of this input to the core. |
| DBGREQ  | is level-sensitive input that causes the core to enter debug state immediately the current instruction has completed execution. This allows external hardware to force the ARM into debug state.  |
| DBGACK  | This is an output from the ARM that goes HIGH when the core is in debug state. This allows other peripherals or the debugging system to determine the current state of the core, and act accordingly.   |

Refer to the **Debug Interface** section of the appropriate ARM data sheet for further details.



## 7.1.2 EmbeddedICE macrocell

The EmbeddedICE macrocell is the integrated on-chip logic that provides debug support for ARM cores. Its presence is indicated by the **I** suffix on the core name.

The EmbeddedICE macrocell is programmed in serial through the *Test Access Port (TAP)* controller on the ARM via the JTAG interface (see **7.9 Reset and JTAG Signal Connection** for details of designing this into your own target). This is usually achieved by using the EmbeddedICE interface (see **7.1.3 EmbeddedICE interface**).

The EmbeddedICE macrocell consists of two real-time watchpoint units, together with a control and status register. One or both watchpoint units can be programmed to halt the execution of instructions by the ARM core via its **BREAKPT** signal. Execution is halted when a match occurs between the values programmed into the EmbeddedICE macrocell and the values currently appearing on the address bus, data bus and various control signals. Any bit can be masked to prevent it from affecting the comparison. Either watchpoint unit can be configured to be a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches).

Please refer to the **EmbeddedICE macrocell** (or **ICEBreaker**) section of the appropriate ARM data sheet for further details.

## 7.1.3 EmbeddedICE interface

The EmbeddedICE interface is a JTAG protocol conversion unit. This translates the debug protocol messages sent out by the debugger into JTAG signals that can be sent to the EmbeddedICE macrocell (and vice versa). The interface can be connected to the host computer using either the built-in serial port or both the built-in serial port and the parallel port. The serial port can be used for bidirectional transfers, but the parallel port can only be used to download code. Using the parallel port increases the maximum download speed with EmbeddedICE to approximately 20KB/s. This is considerably greater than using the serial ports alone (even at 38400 baud), and so is the recommended method of using EmbeddedICE (host permitting).

## 7.1.4 How EmbeddedICE differs from a debug monitor

A debug monitor, such as Angel, is an application that runs on the board in conjunction with the user application, and requires some resource to be available for it on the board. When the board powers up, Angel installs itself by initializing the vector table so that it takes control of the board when an exception occurs. Communication coming in from the host causes an interrupt, halting the user application and calling the appropriate code within Angel. Angel then returns to the user application. This can complicate matters if the application also requires access to interrupts. Similarly, if the application requests some form of I/O to the host, this is implemented within the application using a SWI instruction that is dealt with by Angel's SWI handler (For further details, see **7.8 Semihosting**). This means that Angel requires ROM to store the debug monitor code, RAM to store its data, and control over the exception vectors to allow it to gain control of the ARM while the user application is running.

# EmbeddedICE

---

EmbeddedICE on the other hand requires no such resource. Rather than existing as an application on the board, it works by using a combination of the additional debug hardware on the core and the interface box that handles communication between the core's debug hardware and the host. EmbeddedICE has been designed to allow debugging via the JTAG port to be as non-intrusive as possible:

- The debuggee needs no special hardware to support debugging (the EmbeddedICE macrocell and the JTAG TAP controller are all that is required).
- No memory in the debuggee system need be set aside for debugging, and no special software need be incorporated to allow debugging.
- Execution of the debuggee should only be halted when a breakpoint or watchpoint has been hit or the user requests that the debuggee is halted.

**Note 1** *Although EmbeddedICE requires no memory on the target to operate, the target still requires some memory for executing the application code.*

**Note 2** *EmbeddedICE may pollute the exception vectors if semihosting and/or vector catch are enabled. (See **7.7 Vector Breakpoints and Exceptions**, and **7.8 Semihosting**.)*

## 7.2 The Effect of EmbeddedICE on the Debuggee

Although EmbeddedICE is generally non-intrusive, there are two exceptions:

- When an ARM debugger is started up it attempts to find out the state of the debuggee. To do this, it halts the debuggee and inspects the state of the ARM registers. This, however, can be considered non-intrusive if the debugging session is started after the debugger has been started.
- Watchpoints on structures or arrays larger than one word may cause the debuggee to halt execution when writes occur close to the watchpointed area. EmbeddedICE will restart execution transparently to the user, but this may still cause problems if the application is real-time. For more information see **7.5 Configuring the Debugger** on page 7-9.

# EmbeddedICE

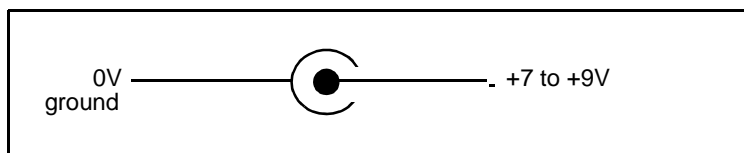
## 7.3 Connecting and Powering Up

To connect and power up the EmbeddedICE interface:

- 1 Connect a serial cable between the 9-pin serial port of EmbeddedICE and the serial port of the host computer.
- 2 Optionally, connect a 25-way D-type parallel cable between the EmbeddedICE and the host computer's parallel printer port.
- 3 Power up the target board as detailed in the appropriate documentation.
- 4 Connect the EmbeddedICE interface to an external +7 to +9V DC (unregulated) power supply, 500mA or greater.

Note that the interface has diode protection to prevent reversed supplies from damaging the board. For correct operation, the 2.1mm power connector should have the positive supply connected to the center pin, as shown in **Figure 7-1:**

**Power connection:**



**Figure 7-1: Power connection**

- 5 Power up the EmbeddedICE interface.
- 6 Connect the EmbeddedICE interface to the target board using a 14-way IDC JTAG cable. (The actual location of the JTAG connection on the target board depends on the board you are using. Refer to the documentation that accompanied the board for details.)

## 7.4 Configuring EmbeddedICE to Match your Target Core

The way in which the EmbeddedICE interface accesses your target hardware is controlled by the software image, or *agent*, in the built-in ROM. This software contains configuration files to allow EmbeddedICE to talk to various ARM cores. Typically these are:

ARM7DI	ARM7 core with debug extensions and EmbeddedICE macrocell (includes ARM7DMI)
ARM7TDI	ARM7 core with Thumb and debug extensions and EmbeddedICE macrocell (includes ARM7TDMI)

When you start EmbeddedICE, it automatically defaults to a particular configuration according to which version of the agent is installed:

- versions up to 1.03 default to ARM7DI
- versions after this default to ARM7TDI

You can usually ignore this message if it is displayed when you start up the debugger (or reload an image):

```
Error during initialization:
Recoverable error in RDI initialization
```

**Note** *Always ensure the selected configuration is correct, otherwise you may have problems when you try to run the image.*

If the following message is displayed, the wrong configuration is being used, and EmbeddedICE has failed to synchronize with the target:

```
Target processor not stopped
```

You can change the current configuration easily from within the debugger. See **7.5 Configuring the Debugger** for details.

On ARM7DI, using the ARM7TDI configuration can lead to breakpoints not being hit. If you press ^C (using UNIX armsd) or the stop button (using the ARM Debugger for Windows) to interrupt the execution, you enter the undefined instruction trap.

On ARM7TDI using the ARM7DI configuration can result in a blank execution window in ADW.

### 7.4.1 About the agent

At the time of writing, the following versions of the agent have been released. The main differences between these are also given.

- |      |  |
|------|--|
| 1.00 | Initial release.   |
| 1.01 | Increased download speed and stepping speed.<br>Full support for serial comms at 19200 and 38400 baud. |
| 1.02 | Support for Thumb-aware cores, with a built-in 7TDI configuration file.                                |
| 1.03 | 26-bit support added.  |
| 1.04 | Support for THUMB Rev 1 silicon.   |

Full support for big-endian targets.

- 1.05 Fixes a problem with 1.04 on ARM7TDMI silicon, where interrupts were not re-enabled when leaving debug state.
- 1.06 Works around a problem with ARM7TDMI when the user forces execution to halt.
- 1.07 Works with slowed JTAG port driver—suitable for use with 2v ports.
- 2.00 ADP version of Embedded ICE which supports debugging applications that use the Angel SWIs for semihosting. It is not compatible with applications that use Demon SWIs for semihosting.
- 2.01 Works around a problem with ARM7TDMI when the user forces execution to halt.
- 2.02 Works with slowed JTAG port driver—suitable for use with 2v ports.

**Note** *All versions lower than 2.00 use RDP, and these are compatible with programs that use Demon SWIs for semihosting (not Angel SWIs). In addition, different command-line arguments are required to specify RDP and ADP, see Application Note 39: Demon and RDP (ARM DAI 0039) for more details.*

If you have problems with your current version, you are advised to contact your supplier and obtain the latest version of the agent.

## 7.4.2 Using a replacement version of the agent

You can use a replacement version of the agent in either of the following ways:

- Program a new EPROM (AM27C010-120DC or equivalent) with this image, and replace the existing EPROM on the lower of the two circuit boards in the EmbeddedICE interface (taking care to reassemble the two boards correctly).
- Download the new image as the first thing you do during a debug session.

Note that the image must be reloaded whenever you cycle the power or press the reset button on the EmbeddedICE interface. (This can be done from within the debugger using the `loadagent` command. See **7.5 Configuring the Debugger**, for details.)

It is possible to upgrade from an RDP-based version of EmbeddedICE to an ADP version in this manner but, after receiving a message that the New Debug Agent is about to start, the debugger will appear to fail. The debugger must be closed down and restarted with ADP selected. Do not reset the EmbeddedICE board because it will revert to the original RDP version of EmbeddedICE.

**Note** *If a new image is downloaded to the EmbeddedICE unit, any later debugger startup message will state that the ROM CRC has failed. This is because the downloaded image is CRC-checking against the ROM image whose checksum is usually different. This message can therefore be ignored.*

## 7.5 Configuring the Debugger

You can now configure the debuggers to use EmbeddedICE.

### 7.5.1 Configuring armsd to use EmbeddedICE

The following armsd command-line options allow access to EmbeddedICE interface facilities (see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for details):

`-selectconfig name version`

Specifies the target configuration to use.

*name* indicates the configuration to use, typically ARM7DI or ARM7TDI.

*version* indicates which version should be used:

*any* accept any version (default).

*n* must use version *n*.

*n+* must use version *n* or later.

The highest numbered version meeting the version constraint is used.

`-loadagent file`

Downloads the replacement agent software into EmbeddedICE and starts it in RAM, rather than using the one in ROM. See **7.4 Configuring EmbeddedICE to Match your Target Core** for further details.

`-loadconfig file`

Specifies the file containing configuration data to be loaded. Note that the required configuration data is contained within the agent itself.

The following commands allow access to EmbeddedICE configuration and agent facilities from within armsd:

`listconfig`

Lists the configurations known to the debug agent. Note that the first configuration listed is the default for the version of the agent in use.

`selectconfig name version`

Specifies the target configuration to use.

*name* indicates the configuration to use, typically ARM7DI or ARM7TDI.

*version* indicates which version should be used:

*any* accept any version (default).

*n* must use version *n*.

*n+* must use version *n* or later.

The highest numbered version meeting the version constraint is used.

## `loadagent file`

Downloads the replacement agent software into EmbeddedICE and starts it in RAM, rather than using the one in ROM. See **7.4 Configuring EmbeddedICE to Match your Target Core** for further details.

## `loadconfig file`

Specifies the file containing configuration data to be loaded. Note that this option is seldom needed because the required configuration data is usually contained within the agent itself.

## Specifying endianness

When downloading an executable to a target using EmbeddedICE, it is also necessary to tell `armsd` the endianness of the target. This contrasts with downloading directly to a board running Angel, or using ARMulator, where there is a default endianness (little-endian).

For example, to download the `example` executable to a little-endian target using EmbeddedICE version 2.0 or later, the standard serial and parallel ports and the default linespeed, enter:

```
armsd -li -adp -port s,p example
```

where:

`-li`            The little-endian switch. Use `-bi` for big-endian.

`-adp`           The switch to allow communication using the ARM Debug Protocol required by Angel.

`-port s,p`      Indicates that the standard serial and parallel ports are to be used.

`example`       The filename.

If you want to use serial port 2 (rather than serial port 1), and linespeed of 38400, enter:

```
armsd -li -adp -port s=2,p -linespeed 38400 example
```

If you are using a version of EmbeddedICE earlier than 2.00 (that uses RDP rather than ADP communication), enter:

```
armsd -rdp -li -serpar example
```

or for serial port 2 (rather than serial port 1), and linespeed of 38400, enter:

```
armsd -rdp -li -serpar -port 2 -linespeed 38400 example
```



## 7.5.2 Configuring the ARM Debugger for Windows to use EmbeddedICE

The *ARM Debugger for Windows (ADW)* can be configured to access a remote target rather than the ARMulator. To do this:

- 1 Choose **Configure Debugger** from the **Options** menu. The Configure Debugger dialog is displayed.
- 2 Click on the **Target** tab.



- 3 Select **remote\_a** or **remote\_d** (use **Add** to select **remote\_d**) from **Target Environment**, ensuring that it is correctly configured. Refer to the *ARM Software Development Toolkit Reference Guide (ARM DUI 0041)* for more details on the target environments. See **3.7 Debugger Configuration** on page 3-28 for more information on the debugger configuration.

**Note** *Angel will time out and default back to ARMulator if the target board does not respond within approximately five seconds.*

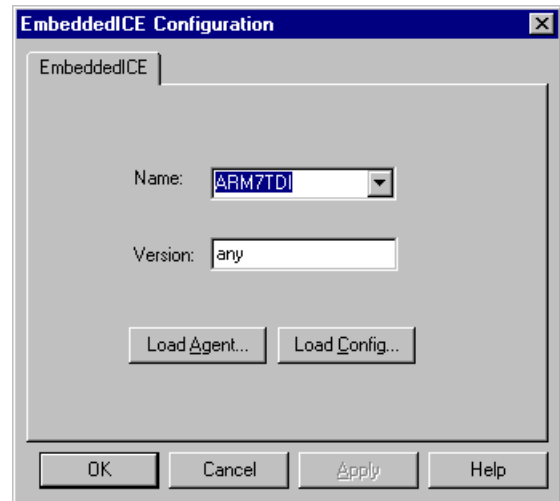
- 4 Select the serial line speed and communications port.
- 5 Click on **OK**.

The debugger is restarted.

## Configuring the EmbeddedICE interface to match the target core

If you need to configure the EmbeddedICE interface to match the target core:

- 1 Choose **Configure EmbeddedICE** from the Options menu. The following dialog is displayed:



- 2 Do one of the following:
  - Choose the name from the list and type in the version. You can usually type **any** for the version. Click the **Select** button.
  - Load a new agent: click the **Load Agent** button, select the required agent file, and then click the **Open** button.
- 3 Click **OK**.

If an image has not been loaded, you can load now by choosing **Load Image** from the File menu.

If the target board being used runs in big-endian mode, you must tell the debugger this when it loads the image (as the default is little-endian mode). To do this:

- 1 Choose **Debugger Configuration** and the **Debugger Tab**.
- 2 Click on **-bi** in the endian selection.

When the debugger closes down, it remembers the remote debugging option settings, so you can automatically make use of them again next time you use the debugger.

## 7.6 Watchpoints and Breakpoints

As with ARMulator, the ARM debuggers provide break, watch, unbreak and unwatch facilities with EmbeddedICE linked targets. However, you should be aware of the following differences.

### 7.6.1 Watchpoints

All ARM debugger watchpoints are data-changed watchpoints, that is, they are not activated if the data point is read or written to with the same data value as the one currently in memory.

See **7.11 Accessing the EmbeddedICE Macrocell Directly** for details of how to implement other forms of watchpoint.

### 7.6.2 Inspecting points

When you inspect the current breakpoints and watchpoints (using the `watch` or `break` commands without arguments within `armsd`, or by choosing **Breakpoints** or **Watchpoints** from the View menu within the ADW), the output specifies whether they are hardware or software points.

### 7.6.3 Hardware versus software points

Hardware breakpoints are implemented using an EmbeddedICE macrocell point to spot an instruction fetch from the appropriate address. This works in all cases, even if the program being debugged modifies itself as it executes, or if the code is in ROM. However, it completely ties up one EmbeddedICE macrocell point.

Software breakpoints are implemented using an EmbeddedICE macrocell point to spot an instruction fetch of a particular bit-pattern. This bit-pattern will have been previously stored at the appropriate location, and the real instruction noted. Therefore, self-modifying code or code in ROM cannot be debugged using this type of breakpoint. (EmbeddedICE will not attempt to use software breakpoints for code in ROM anyway.) Any number of software breakpoints can be supported using a single EmbeddedICE macrocell point.

Hardware watchpoints are implemented using an EmbeddedICE macrocell point to spot data writes to addresses that fall inside a mask. This type of watchpoint is efficient, as execution stops only when the relevant data is written, but it ties up an EmbeddedICE macrocell point completely. Note also that if a structure or an array is being watchpointed, the mask is likely to include some addresses that are not part of the object being watchpointed. In this case, writes to these unwanted addresses are filtered out by EmbeddedICE interface. Execution performance is slightly degraded, because the processor is stopped when the unwanted watchpoint is hit, and then restarted automatically by the EmbeddedICE interface.

Software watchpoints make no use of the EmbeddedICE macrocell. Instead, after each instruction is executed, the data locations concerned are examined to see whether their values have changed. If a value has changed, execution is halted; otherwise execution is

resumed. This type of watchpoint reduces execution performance drastically. In addition, it clearly cannot be used on write-only areas of memory, such as some memory-mapped device registers.

## 7.6.4 Watchpoints, breakpoints and the program counter

Watchpoints are taken when the data being watchpointed has changed. When this happens, the program counter is updated to point to the instruction following the one that caused the watchpoint to be taken. The value of the watchpointed data is therefore the new value, not the old value.

Breakpoints are taken when the instruction being breakpointed reaches the execution stage of the pipeline, but before it is executed. So, when the breakpoint is taken, the program counter is not updated and retains the address of the breakpointed instruction.

**Note** *Inside the core of an ARM, the program counter typically points two instructions beyond the currently executing instruction (historically this is the address of the instruction currently being loaded into the fetch stage of the pipeline). However, the ARM debuggers simplify this by reporting a modified value for the program counter, so that when it is displayed within the debugger, its contents are the address of the instruction being (or about to be) executed.*

7.7 Vector Breakpoints and Exceptions

When the debugger starts executing the target application, EmbeddedICE puts into place any breakpoints specified by the debugger internal variable `$vector_catch`. (This means that when you start executing, user breakpoints and watchpoints may have to be downgraded from hardware ones to software ones without warning.) The `$vector_catch` variable is used to indicate whether or not execution should be trapped when various conditions arise. The default value is `%RUsPDAiFE`, where capital letters indicate that the condition is to be intercepted.

Breakpoint	Description
R	Reset
U	Undefined instruction
S	SWI
P	Prefetch abort
D	Data abort
A	Address exception
I	IRQ
F	FIQ
E	Error (reserved for possible, future software fault detection)

Table 7-1: Breakpoints

This is useful if the application contains no specific handler for a particular exception.

In normal usage, the SWI flag within `$vector_catch` remains lowercase, as finer control is provided by the debugger internal variables `$semihosting_enabled` and `$semihosting_vector` (see **7.8 Semihosting** for further details).

In the case of a system with no interrupt handler that has an active source of interrupts, set up the exception vectors to mask out all the interrupts by loading the instructions shown below.

You can do this from the `armsd` prompt using the following commands. You do not need to type the text shown after the semicolon (;) as these are comments showing the instruction encoded in each hexadecimal value.

```
0x18 = 0xE1A00000; NOP
0x1C = 0xE14FD000; MRS r13, spsr
0x20 = 0xE38DD0C0; ORR r13, r13, 0xC0
0x24 = 0xE169F00D; MSR spsr, r13
0x28 = 0xE25EF004; SUBS pc, lr, #4
```

You can simplify the above by placing the commands in an `armsd.ini` startup file. All the commands are then executed when `armsd` is invoked. For information on setting up an `armsd` startup file, refer to the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

If you are using ADW, choose **Memory** from the **View** menu and then modify the contents of the appropriate addresses.

There may be slight problems with applications that rely on interrupts occurring in real time. When doing any operation via EmbeddedICE other than `go` (for example hitting a breakpoint), interrupts will occur during the operation, as going into debug state takes time. If this causes major problems, the workaround is to turn off interrupts (either patch the start-up code, or manually set the CPSR). Then, when you require an interrupt, use the debugger to enter the correct mode, set `pc` equal to vector address, and restart execution.

## 7.8 Semihosting

Semihosting is a mechanism whereby the ARM target communicates I/O requests made in the application code, such as those provided in the standard ANSI C library (eg. `printf()` calls to the screen or calls to `scanf()` from the keyboard) up to the host system, rather than having a screen/keyboard/disk on the target system itself.

On target boards containing either Demon or Angel, this is implemented using a set of defined SWIs. Thus, Demon or Angel installs a SWI handler when the board is powered up, and, when the target executes a SWI instruction, Demon or Angel carries out the required communication with the host.

**Note** *Demon and Angel use different, incompatible, SWI mechanisms. See **Chapter 5, Angel** for more details.*

When using EmbeddedICE, semihosting is handled differently. It is implemented by “faking” the SWI handler that Demon or Angel would have installed on the SWI vector.

EmbeddedICE installs a breakpoint on the appropriate vector, and when this breakpoint is hit, checks to see what the SWI was. If the SWI is recognized, EmbeddedICE emulates it and restarts execution of the application transparently. If the SWI is not recognized as a semihosting one, EmbeddedICE halts the processor and reports an error.

**Note** *EmbeddedICE versions before 2.00 emulate Demon SWIs only, whereas versions 2.00 onwards emulate Angel SWIs only.*

This semihosting via EmbeddedICE can be disabled or changed by making use of the following debugger internal variables:

`$semihosting_enabled`

By default, this variable is set to 1 to enable semihosting. Setting it to 0 disables semihosting. This can be useful, for example, when debugging applications running from ROM. Disabling semihosting in such situations frees up another watchpoint unit.

`$semihosting_vector`

This variable controls the location of the breakpoint set by EmbeddedICE to detect a semihosted SWI. It is set to 8 by default. Note that EmbeddedICE will return directly to the instruction following the SWI instruction in your code after handling the semihosted SWI, completely bypassing the contents of the `$semihosting_vector` address. (This is done by examining the contents of `1r`.) Note that if this variable is set to zero, this does not imply address 0. Address 8 is used instead. In addition, all exceptions and interrupts are trapped and reported as an error condition, no matter what the value of `$vector_catch`.

In the ADW, both of these variables can be accessed by selecting **Debugger Internals** from the **View** menu.

## 7.8.1 Adding an application SWI handler when using EmbeddedICE

In addition to using semihosted SWIs, many applications will also need to install their own SWI handlers into the vector table. This must be done in such a way that the application SWI handler will successfully cooperate with the EmbeddedICE semihosting mechanism. To do this, the application SWI handler must be installed into the vector table. Then the `$semihosting_vector` must be modified to point to a location at the end of the application handler that is only reached if your handler does not recognize the SWI (or recognizes it as a semihosting SWI).

It is essential that the actual position within the application handler to which the `$semihosting_vector` points is correct. Both `lr` and `spsr` must have been restored to the values they contained when the SWI exception was taken. If this is not done, the EmbeddedICE cannot return correctly to the application after handling the semihosted SWI. All other registers should also be restored. Typically, this means that they are restored from the stack.

For example, a particular SWI handler may detect if it has failed to handle a SWI and branch to an error handler (see **Chapter 10, Exception Handling**, for further details of writing SWI handlers):

```
                                ; r0 = 1 if SWI handled
CMP r0, #1                    ; Test if SWI has been handled.
BNE NoSuchSWI                 ; Call unknown SWI handler and quit.
LDMFD sp!, {r0}               ; Unstack SPSR...
MSR spsr, r0                  ; ...and restore it.
LDMFD sp!, {r0-r12,pc}^       ; Restore registers and return.
```

This code could be modified for use in conjunction with EmbeddedICE semihosting as follows:

```
                                ; r0 = 1 if SWI handled
CMP r0, #1                    ; Test if SWI has been handled.
LDMFD sp!, {r0}               ; Unstack SPSR...
MSR spsr, r0                  ; ...and restore it.
LDMFD sp!, {r0-r12,lr}        ; Restore registers.
MOVEQS pc, lr                 ; Return if SWI handled.
Semi_SWI
    MOVS pc,lr                 ; Fall through to EmbeddedICE handler
```

`$semihosting_vector` should then be set up to point to the address of `Semi_SWI`. Note that the instruction at this address never gets executed because EmbeddedICE returns directly to the application after processing the semihosted SWI. However, using a normal SWI return instruction ensures that the application does not crash if the semihosting breakpoint is not set up. The semihosting action requested is simply not carried out and the handler just returns.



There is one slight complication if the application is linked with the ARM C library, and hence uses the C library's startup code. If `$semihosting_vector` is set to the "fall through" part of application SWI handler before the application starts execution, the semihosted SWIs that are called by the library startup (eg. the SWI to get heap and stack information) triggers an unknown watchpoint error.

This is because the SWI vector at this point has not yet had the application handler installed into it, and may still contain the software breakpoint bit pattern. This triggers a breakpoint that EmbeddedICE no longer knows about because the `$semihosting_vector` address has moved to a place that cannot currently be reached. To prevent this from happening, change the contents of `$semihosting_vector` just before the application installs its own handler, typically by setting a breakpoint in the main code.

**Note** *If semihosting is not required at all by an application, this process can be simplified: all that is needed is to set `$semihosting_enabled` to 0.*

Care is therefore required when moving an application that previously ran in conjunction with Demon or Angel onto an EmbeddedICE system. On Demon- or Angel-based systems, application SWI handlers are typically added by moving (and adjusting) the contents of the SWI vector (as installed by Demon or Angel) to another place, and installing the application SWI handler into the SWI vector. Such a method will not work correctly under EmbeddedICE as there is no instruction to move out of the SWI vector. So when the application handler fails to handle a particular SWI, it will jump to a storage location and try to execute a completely random instruction, that will typically be undefined. Thus it is essential when moving an application onto an EmbeddedICE-based system to convert to the correct way of installing the application and semihosted SWI handlers.

## 7.9 Reset and JTAG Signal Connection

When you design a target system, it is important to consider the connection of the JTAG and reset signals. The different approaches that may be used are described here.

**nRESET** is used to reset the processor core and put it into a known state, while **nTRST** is used to reset the TAP controller and the EmbeddedICE macrocell, including the registers in the breakpoint/watchpoint units. Both these resets must be applied before the device will function correctly.

The system must be designed so that it operates correctly whether or not the EmbeddedICE interface is connected. It must also allow the EmbeddedICE interface to reset the JTAG TAP and, in some cases, the system itself.

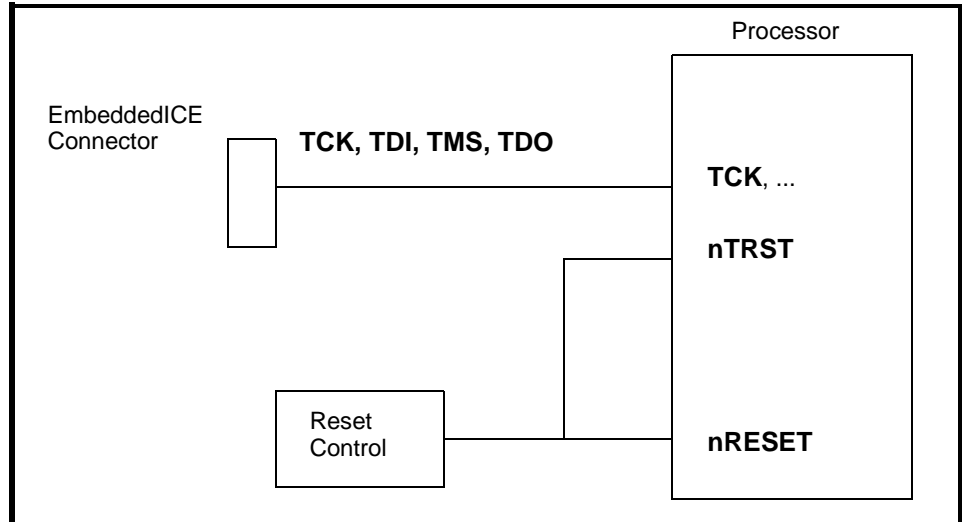
The following sections describe two different approaches to the generation of the **nTRST** signal. The first approach is the simplest and provides a basic reset scheme, whereas the second approach is more sophisticated and allows a greater level of control by the debugger.

### 7.9.1 Basic nTRST connection

As the system must function correctly when the EmbeddedICE interface is not connected, the following must occur before the system can operate:

- **nRESET** must be held LOW for a number of clock cycles, before going HIGH and allowing the processor out of reset
- **nTRST** must be held LOW to ensure the TAP is reset, and this can be done in either of the following ways:
  - by tying **nTRST** permanently LOW
  - by holding **nTRST** LOW, before going HIGH

However, if **nTRST** is permanently tied LOW, debugging via the JTAG port cannot occur, so **nTRST** must be pulsed LOW. The simplest way to provide this is for **nTRST** to be connected directly to **nRESET**.



**Figure 7-2: Basic nTRST connection**

The EmbeddedICE unit is still able to reset the TAP, because it is possible to reset the TAP using only a combination of the **TCK** and **TMS** signals.

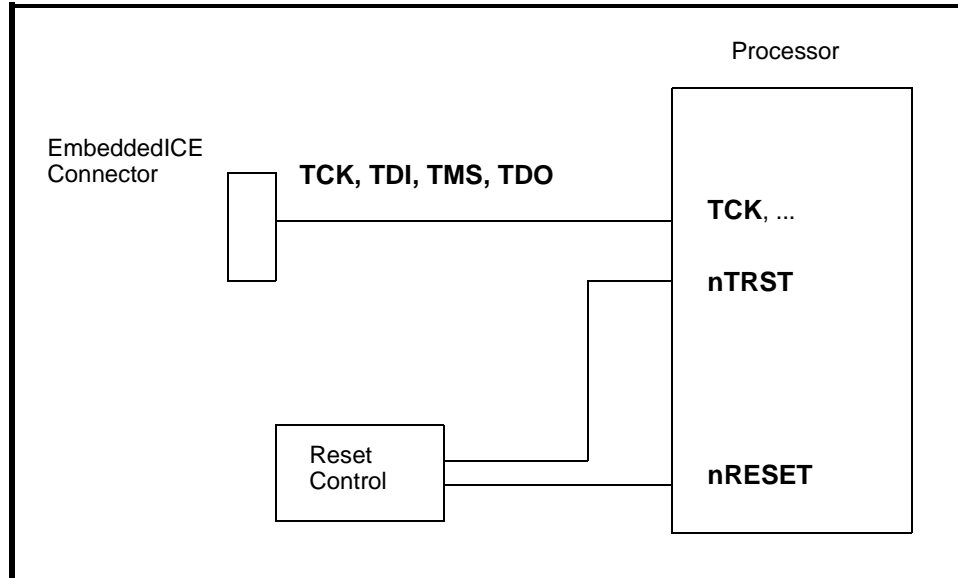
The disadvantage of this approach is that it is not possible to set a breakpoint on the processor exiting from reset. This is because the TAP is always held in reset when the processor is in reset, and the breakpoint registers cannot be programmed while the TAP is in this state.

## 7.9.2 Separate control of nTRST and nRESET

In some situations it is desirable to be able set a breakpoint on the reset vector, so that the target system is stopped when the processor is exiting from reset.

However, to do this, the breakpoint registers must be programmed prior to the processor exiting from reset, and therefore it is necessary to have separate control of the **nTRST** and the **nRESET** signals.

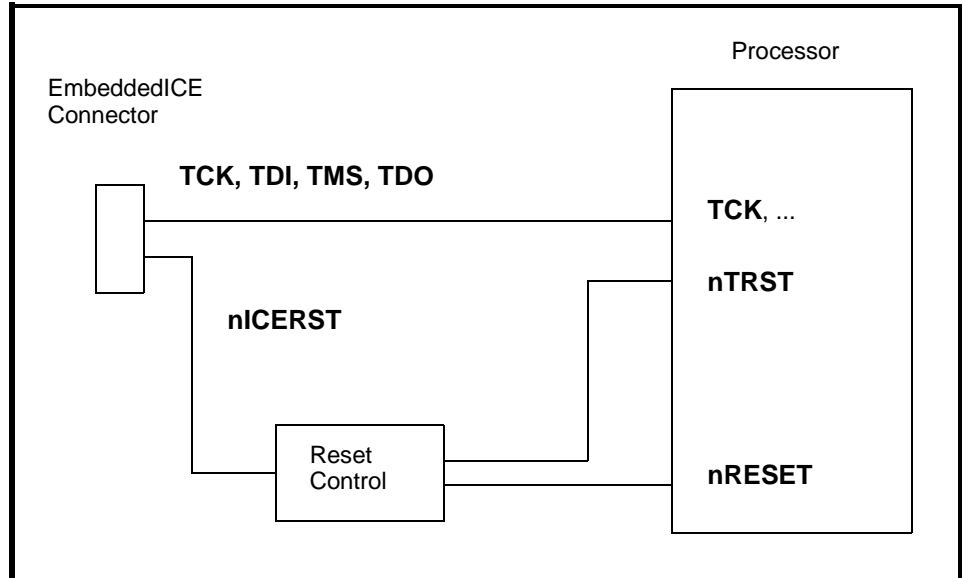
The diagram below shows a suggested connection method. The reset control logic, which may be a simple PLD, is used to gate the resets such that **nTRST** is pulsed LOW at initial power-up only, and **nRESET** is pulsed LOW both at power-up and whenever any other form of reset is asserted.



**Figure 7-3: *nTRST* and *nRESET* connection**

This allows the user to set a reset breakpoint by setting a breakpoint on address 0 and then pressing the reset button. If the basic scheme has been used, as described in **7.9.1 Basic *nTRST* connection**, pressing the reset button also resets the breakpoint registers.

The EmbeddedICE connector also includes an **nICERST** signal that is asserted by the EmbeddedICE unit when it wishes to reset the target system. This signal may be used by the reset controller to remove the need for the user to press the reset button, as shown in **Figure 7-4: *nTRST*, *nRESET* and *nICERST* connection**.



**Figure 7-4: *nTRST*, *nRESET* and *nICERST* connection**

## Notes

- In some EmbeddedICE documentation the **nICERST** signal is called **nSYSRST**.
- The ability to reset the target system from the debugger via EmbeddedICE, rather than having to press a reset button, is not implemented in SDT 2.1.

## 7.10 Debugging Applications in ROM

EmbeddedICE can be used to debug systems running in ROM. Typically, when a target board with an application stored in ROM is powered up, the application begins running. Thus, when the JTAG cable is connected and the debugger started up on the host, the processor is stopped. At this stage, the application could be at any point in its execution lifetime, depending on when EmbeddedICE was connected.

This means that the state of the system can be examined and execution can be restarted from the current place. In some cases this may be sufficient to achieve what is required. However, in many cases it is preferable to restart execution of the application as if from power-on. There are two ways of approaching this:

- by faking the reset
- by carrying out an actual reset of the target

If you have set up a basic **nTRST** connection, as described in **7.9.1 Basic nTRST connection**, you may only fake the reset. If you have implemented additional use of ICE Reset as described in **7.9.2 Separate control of nTRST and nRESET**, you may choose either method.

When you debug code running from ROM, ensure that at least one watchpoint unit remains available to allow breakpoints to be set on code in ROM (as software breakpoints cannot be used). The chances of the debugger taking these units for its own use can be reduced by not using semihosting or trapping on any unusual exceptions. To do this, the following debugger internal variables should be set as soon as possible after starting up the debugger:

```
$semihosting_enabled = 0
$vector_catch = 0
```

Then setup any ROM breakpoints before any non-ROM breakpoints or watchpoints are set, otherwise, again, the watchpoint units may be fully occupied, causing the attempt to set the ROM breakpoint to fail:

```
Error: Too many breakpoints
```

Watchpoint units must then be freed (by deleting breakpoints / watchpoints) before the ROM breakpoints can be set.

Another complication with debugging a system in ROM is that the ROM image cannot contain any debug information. When debugging using EmbeddedICE, symbol or source code information is available by loading the relevant information into the debugger from a file on the host. This is described in **7.10.3 Accessing debug information**.

### 7.10.1 Faking a reset

You can simulate a reset from within the debugger, typically, by setting:

PC to 0	address of the reset vector
CPSR to %IF_SVC32	to change into supervisor mode with interrupts disabled

This simulates the state of the ARM at power on/reset, but it does not allow for a reset memory map or the initialization of any target-specific features (such as registers). If possible, it is therefore necessary to modify any such target-specific features to resemble their startup configuration before executing the application again. You can automate this procedure with the scripting facility (the `obey` command).

## 7.10.2 Carrying out a real reset

In some circumstances, it is possible to carry out a real reset of the board. However, care is required when doing this because if the EmbeddedICE macrocell is also reset, the debugger becomes out of sync with the macrocell. Two forms of reset are required on the board:

- a full power-on reset, that resets everything on the board
- a reset button that resets everything on the board except the EmbeddedICE macrocell

See also **7.9 Reset and JTAG Signal Connection**.

Thus, if a *hardware* breakpoint is set on the reset vector (or the address in ROM of the reset routine to which the reset vector branches), when the target is reset (but *not* the EmbeddedICE macrocell, otherwise the breakpoint will be lost), the target will halt on reset as required.

**Note** *It is advisable to delete all other software breakpoints/watchpoints before resetting, because although they may be deleted by the reset, the debugger will think they still exist. This can cause problems when you try to delete them after the reset.*

### Example: The ARM PIE7 board (with ARM7DI-type silicon fitted)

The ARM7 PIE board implements the required two levels of reset. Unfortunately, the reset switch on the board forces another power-on reset, and hence cannot be used for debugging from reset. The initialization reset only drives **nRESET** (and the other devices on the board), but is intended to be driven by external logic and so needs a de-bounced switch to be added. Signal **XHNINIT**, on pin 16 of the parallel port, can be used to reset the board without affecting the EmbeddedICE macrocell and its breakpoints.

### Example: The ARM PID7T board

THE ARM PID7T board also implements the required two levels of reset. In this case, the reset switch carries out the required initialization reset so allowing debug from reset. All that is required in this case is to set the hardware breakpoint and then press the reset button.

## 7.10.3 Accessing debug information

An application built to be executed out of ROM is usually built using a command of the form:

```
armlink -bin -RO-base addr1 -RW-base addr2 -o image f1.o f2.o
```

where:

- |          |   |
|----------|---|
| -bin     | causes a plain binary image to be generated, suitable for blowing into ROM                              |
| -RO-base | sets the base address of the read-only part of the application (this is equivalent to the -base option) |
| -RW-base | sets the base address used to store data by the application (this is equivalent to the -data option)    |
| -o       | sets the name of the output file generated  |

Such a binary image cannot contain any symbol or source code debug information. However, it is possible to produce the debug information required by EmbeddedICE and load it into the debugger. To do this, you must perform an almost identical second link stage, changing -bin to -bin -aif and giving a different output filename. The output file generated will match the ROM image exactly, but also contains the required debug information.

```
armlink -bin -aif -RO-base addr1 -RW-base addr2 -o debug f1.o f2.o
```

This debug information is either source-level if the files were compiled/assembled with the -g option, or low-level. (armlink provides all available debug information by default; use the -Nodebug link option if no debug tables are to be generated.)

**Note** *This must be a -bin -aif image and not a -aif image, because in the latter all addresses would be offset by 128 bytes compared to the image in ROM, since the base address would specify the start of the header rather than the start of the application code.*

The toolkit provides commands to read the debug information out of the -bin -aif file, without having to download the image as well.

### For armsd

Use one of the following:

```
armsd -symbols filename
```

when invoking the debugger, or:

```
readsyms filename
```

when armsd is running (where filename is the -bin -aif file corresponding to the image in the system ROM).

### For ADW

Choose **Load symbols only** from the **File** menu, and then select the -bin -aif file corresponding to the image in the system ROM.



## 7.10.4 Debugging systems with ROM at zero

When debugging systems with ROM at zero, rather than RAM, it is necessary to set `$vector_catch` to 0. This prevents EmbeddedICE from trying to set software breakpoints on the vector table.

## 7.11 Accessing the EmbeddedICE Macrocell Directly

No new debugger commands have been added to allow you to manipulate EmbeddedICE macrocell registers. Instead, you can use the commands that display and set coprocessor registers, with coprocessor number 0 specified. Coprocessor 0 is defined so that it is never implemented, and therefore cannot clash with user- or ARM-developed coprocessors.

For example, the `armsd` command `cregisters 0` displays the contents of a number of registers that are in fact EmbeddedICE macrocell registers.

```
ARMSD: cregisters 0
c0 =      0x05
c1 =      0x09
c4 =      0x00
c5 = 0x00000000
c8 = 0x516ce8da
c9 = 0xbfdf0ea6
c10 = 0xbff6fd7d
c11 = 0xfbaffbfff
c12 =      0x0000
c13 =      0xff
c16 = 0x00000008
c17 = 0x00000003
c18 = 0x7dfeeffb
c19 = 0xffffffff
c20 =      0x0100
c21 =      0xf6
```

To access the coprocessor 0 registers using ADW, choose **Registers** from the View menu, and then display the Coprocessor dialog box. Enter 0 for the coprocessor number and check the Raw (unformatted) display option.

The correspondence between coprocessor 0 registers displayed and EmbeddedICE macrocell registers is simple; the `register address` field in the EmbeddedICE macrocell scan chain *is* the register number. For more information about the EmbeddedICE macrocell, refer to one of the ARM data sheets on an ARM core with debug capabilities (eg. ARM7DI, ARM7TDMI).

You may read EmbeddedICE macrocell registers freely in this manner, but writing them needs more care. This is because EmbeddedICE also makes use of EmbeddedICE macrocell registers to set up breakpoints and watchpoints. When you write to an EmbeddedICE macrocell register (for example using the `armsd` command `cwrite 0 20 0x44`), EmbeddedICE checks to see if the breakpoint is in use, of which that register is a part. If it is, EmbeddedICE attempts to free it (by degrading hardware breakpoints to software breakpoints), and then sets a lock on the breakpoint so that EmbeddedICE makes no further attempt to use it.

It is possible to see which breakpoints have been locked in this way by displaying the value of `$icebreaker_lockedpoints`. This debugger internal variable can also be set to unlock breakpoints. In the ARM7DI and ARM7TDI, the breakpoints are numbered 1 and 2, and bits 1 and 2 in `$icebreaker_lockedpoints` indicate their status.

If a breakpoint or watchpoint that has been set up in this way gets taken, EmbeddedICE will not know why execution has stopped (since it was not due to one of the break/watch points it knows about), and so will halt debuggee execution with the report `Unknown watchpoint`.

Note that a debugger hardware watchpoint should not be used in combination with a user-programmed EmbeddedICE macrocell point, because EmbeddedICE will not halt execution due to the user-programmed point. In practice, this is unlikely to cause any problems as the EmbeddedICE macrocell has only two watchpoints, and you can program the second watchpoint directly as well.

Take care when writing EmbeddedICE macrocell registers 0 and 1, the control and status registers. EmbeddedICE uses these to perform many of its operations; if they are written at all, they should always be returned to their original values afterwards.

Note also that debugger requests to read or write EmbeddedICE macrocell registers do not necessarily cause the registers to be read or written immediately. This is because, in the interests of efficiency, the EmbeddedICE software caches the contents of the EmbeddedICE macrocell registers, only updating changed registers before execution of the debuggee is resumed.

## 7.12 EmbeddedICE and Target Board Memory Layout

### Versions 2.00 onwards

Before version 2.00, the EmbeddedICE software assumed it could measure the size of memory by checking bytes from 0 until it found non-RW memory. Versions 2.00 onwards do not make this assumption.

Instead, a debugger variable `$top_of_memory` is read by EmbeddedICE and the value of this is returned by the Angel `heap_info` SWI that is used by the C library for initialization.

The default value for `$top_of_memory` is 512KB, which is correct for ARM7PIE and ARM7T PID boards without expansion RAM. If you have expanded RAM, or are using a different board with more or less RAM, set `$top_of_memory` to the correct value before you run the application.

The mechanism places the application heap above the application program, and the application stack at the top of memory. It also requires that memory is contiguous from the end of the application to the top of memory.

If this is not the case, it is possible to define the following symbols in the application to override this mechanism, and set up the application heap and stack precisely:

```
__heap_base, __heap_limit, __stack_base, __stack_limit.
```

For example, in ARM Assembler

```
EXPORT __heap_base
__heap_base DCD my_heapbase_value
```

### Versions before 2.00

Before version 2.00, the EmbeddedICE software measured memory size by checking bytes from 0 until it found non-RW memory. This assumes that the memory layout is like the PIE60 or PIE7 board.

The top-of-memory address found in this way is returned when the user application does a `SWI_GetEnv` call. This call is made by the ARM C library in the file `kernel.s` at label `__entry`, to establish the correct place to locate the stack before starting execution of a user program. This causes problems if the target board being used does not have RAM in a contiguous block starting at location zero. For instance, a PIV board has two blocks of discontinuous RAM:

0x000000–0x080000	32-bit memory
0x400000–0x440000	16-bit memory

If an application (linked with the C library) is built to run in the 16-bit memory, `SWI_GetEnv` returns 0x080000 as the heap top, the heap base immediately follows the RW data for the application, and so is 0x4xxxxx. When the heap top is less than the heap base, the C library gets very confused.

You can avoid this in one of the following ways:

- Modify `kernel.s` so that it does not call `SWI_GetEnv`, set up the required value manually in the code, and rebuild the C library.
- Do not link directly with the C library. Create your own startup and initialization code, and pull in any required routines from the library. This is described in **Chapter 12, Writing Code for ROM**.

## 7.12.1 Privileged modes and stack pointers

It should be noted that EmbeddedICE itself does not set up any stack pointers. Thus, if you have linked with the full semihosted ANSI C library, you will have a user mode stack pointer setup for you (as described in **7.12 EmbeddedICE and Target Board Memory Layout** on page 7-30). However you will not have stack pointers setup for the privileged modes. This contrasts with the situation where an application is run under ARMulator, Angel or Demon where the stacks are set up for you.

You must therefore ensure that your application code sets up the stack pointers for any privileged modes that may be entered as a result of exceptions taking place.

If you are not linking with the full semihosted ANSI C library, not even the user mode stack pointer will be set up. For details of stack initialization, see **Chapter 13, Writing code for ROM**.

## 7.13 Timer Accuracy

When using EmbeddedICE, the standard ANSI `clock()` function is inaccurate. For example, the following figures have been obtained by running Dhrystone on a PIE7 board and on the PIV7T board, with communication provided by EmbeddedICE.

This was compiled using:

```
armcc -Otime -DMSC_CLOCK dhry_1.c dhry_2.c -o dhry
```

and the results obtained are shown in **Table 7-2: Timings**:

Iterations	PIE7 (Dhrystones)	PIV7T (Dhrystones)
50000	16666.7	16666.7
100000	16666.7	14285.7
200000	18181.8	15384.6

Table 7-2: Timings

The number of Dhrystones produced by each board should be almost identical for invocations with different numbers of iterations, but there will be a difference between the results from the two boards, because of the different board designs.

However, as well as changing dramatically between invocations with different iteration values, the values actually obtained can also fluctuate wildly over several runs. For instance, on some invocations on the PIE7 board, 50 000 iterations produced a figure of 25 000 Dhrystones. Figures may also vary according to which version of the toolkit is being used.

There are two reasons for this:

- The implementation of `SWI_CLOCK` (called by the standard ANSI `clock()` routine) in the EmbeddedICE agent software has a resolution of one second rather than one centisecond. This is despite the fact the value of `CLK_TCK` (that gives the number of clock ticks per second) being 100.
- Currently, the EmbeddedICE agent does not produce the timing figure itself, but rather sends a request back to the host for it. This means that the figure returned is larger than it actually should be, because it includes the time it takes for EmbeddedICE to send a request to the host and get the time back.

Combined, these two issues make any benchmarks (such as Dhrystone) run on systems using EmbeddedICE inaccurate. Extreme care is therefore needed in such cases. Future versions of the agent software may change the way that EmbeddedICE produces timing figures, but there are measures that can be taken to improve matters now. If the application is modified to use a clock routine that takes its values from a timer routine running on the board itself, more reliable figures can be generated. This is explained below in **7.13.1 Modifying Dhrystone**.



## 7.13.1 Modifying Dhrystone

To use a board-specific timer, it necessary to carry out minor modifications to `dhry_1.c` and `dhry.h`.

- 1 In `dhry.h`:
  - Copy the `#ifdef MSC_CLOCK .... #endif` section, changing the `MSC_CLOCK` to `INT_CLOCK`.
  - Within all these copied sections, change the `#include <time.h>` to `#include "time.h"`.
- 2 In `dhry_1.c`:
  - Copy all three of the `#ifdef MSC_CLOCK .... #endif` sections, changing the `MSC_CLOCK` to `INT_CLOCK`.
  - Within all these copied sections, change the reference to `clock()` to `int_clock()`.
  - Add a call to the `init_timer()` routine within `main()` within a new `#ifdef INT_CLOCK .... #endif` section at the point just before the first calls to `printf()`.
- 3 Create a new `time.h` file:
 

```
/*
 * time.h
 */
#define CLK_TCK          100
typedef unsigned int clock_t;
```
- 4 Lastly, create a `time.c` file specific to the target board being used. This needs to contain the timer initialization, timer interrupt and clock routines, plus any necessary constants. Two example `time.c` files are given below, one for the PIE7 board and one for the PIV board (which can be easily modified to work on other RPS-compliant boards, such as the PID7T).

### Version for PIE7 board

```
/*
 * time.c
 * PIE7 board version
 */
typedef volatile unsigned *IOptr ;
#define UartBase    0x80000000

/*
 * Uart registers
```

```
*/
#define CR ((IOptr)(UartBase + 0x08))
#define ACR ((IOptr)(UartBase + 0x10))
#define ISR((IOptr)(UartBase + 0x14))
#define CTUR ((IOptr)(UartBase + 0x18))
#define CTLR ((IOptr)(UartBase + 0x1c))
#define IMR ((IOptr)(UartBase + 0x14))

/*
 * Uart register values required for timer
 */
#define Start_Counter_Timer0x80
#define Timer_Crystal_div160x70
#define Counter_Ready0x10
#define Reset_Counter_Timer0x90
#define Counter_Ready0x10
#define Power_Down0x8

/*
 * TimerTick value required for CTUR and CTLR
 * This calculated from
 * ClockFreq = 3686400; from 3.6864 Mhz external clock
 * TimerFreq = ClockFreq / 16 / 2 ; frequency of the timer
 * TimerTick = TimerFreq / 100 ; 100 times per sec
 */
#define TimerTick 1152

/*
 * Location to store address of exception handlers
 */
#define SoftVectors0xA40
/*
 * Interrupt used for timer
 */
#define INTERRUPT0x1c
/*
 * Counter variable to keep track of clock ticks
 */
```



```
volatile unsigned TimerVal;

/*
 * Timer interrupt handler
 */
__irq static void TimerIncrement (void)
{
    /* Check if timer has ticked */
    if (*ISR & Counter_Ready)
    {
        /* Increment timer */
        TimerVal++;
        /* Reset Timer */
        *CR = Reset_Counter_Timer;
    }
}

void init_timer (void)
{
    *ACR = Power_Down | Timer_Crystal_div16;
    *CTUR = TimerTick / 256;
    *CTLR = TimerTick % 256;
    *(IOptr)(SoftVectors + INTERRUPT) =
        (volatile unsigned)TimerIncrement;
    *(IOptr)INTERRUPT = 0xe59ff000 + SoftVectors - 8;
    *CR = Start_Counter_Timer;
    *IMR = Counter_Ready ;
}

unsigned int_clock (void)
{
    return (TimerVal);
}
```

## Version for PIV board

```
/*
 * time.c
 * PIV board version
 */
typedef volatile unsigned *IOptr ;
```

```
/*
 * PIV board specifics
 */
#define ICBASE 0x1800000
/* Interrupt controller base address */
#define CTBASE0x1840000
/* Counter / timer base address */
#define SYSTEM_CLOCK10000000/* MCLK = 10MHz */

/*
 * RPS values
 */
#define IRQDisable ((IOptr)(ICBASE + 0x0c))
#define FIQDisable ((IOptr)(ICBASE + 0x10c))
#define IRQStatus((IOptr)(ICBASE + 0x00))
#define IRQRawStatus((IOptr)(ICBASE + 0x04))
#define IRQEnable((IOptr)(ICBASE + 0x08))
#define Timer1Load((IOptr)(CTBASE + 0x00))
#define Timer1Control((IOptr)(CTBASE + 0x08))
#define Timer1Clear((IOptr)(CTBASE + 0x0C))
#define Timer2Control((IOptr)(CTBASE + 0x28))
#define Timer2Clear((IOptr)(CTBASE + 0x2C))
#define Timer_Periodic0x40
#define Timer_Enable0x80
#define Timer1_Mask0x10
#define Timer1_Prescale0x04
/* Timer1 Control value for /16 prescale */
#define Timer1_Freq(SYSTEM_CLOCK/16)
#define Timer1_Tick(Timer1_Freq/100)

/*
 * Location to store address of exception handlers
 */
#define SoftVectors0xA40
/*
 * Interrupt used for timer
 */
#define INTERRUPT0x18
/*
```

```

        * Counter variable to keep track of clock ticks
        */
volatile unsigned TimerVal;

/*
 * Timer interrupt handler
 */
__irq static void TimerIncrement(void)
{
    /* Check if timer has ticked */
    if (*IRQStatus & Timer1_Mask)
    {
        /* Increment timer */
        TimerVal++;
        /* Reset Timer */
        *Timer1Clear = 0;
    }
}

void init_timer(void)
{
    *IRQDisable = ~0;
    *FIQDisable = ~0;
    *Timer1Control = 0;
    *Timer2Control = 0;
    *Timer1Clear = 0;
    *Timer2Clear = 0;
    *(IOptr)(SoftVectors + INTERRUPT) = (volatile
                                         unsigned)TimerIncrement;
    *(IOptr)INTERRUPT = 0xe59ff000 + SoftVectors - 8;
    *Timer1Load = Timer1_Tick;
    *IRQEnable = Timer1_Mask;
    *Timer1Control = Timer1_Prescale + Timer_Periodic+Timer_Enable;
}

unsigned int_clock (void)
{
    return (TimerVal);
}

```

## 7.13.2 Building the executable

When you have made the additions and changes to `dhry.h` and `dhry_1.c` (as described in **7.13.1 Modifying Dhrystone**), created `time.h` and the appropriate `time.c`, you can now build the executable. Unless you are targeting a Thumb-compatible ARM processor (see below), enter the following:

```
armcc -c -Otime -DINT_CLOCK dhry_1.c -o dhry_1.o
armcc -c -Otime -DINT_CLOCK dhry_2.c -o dhry_2.o
armcc -c -Otime time.c -o time.o
armlink -o intdhry dhry_1.o dhry_2.o time.o armlib.32l
```

If `armlib.32l` is not in the current directory, you must specify the full pathname.

### Targeting a Thumb-compatible ARM processor

If you need the benchmark to be run as a Thumb application, you can:

- Rebuild the application using `tcc` for the main application code, but still use `armcc` for `time.c`. This is necessary because the interrupt handler contained in `time.c` must be compiled into ARM code.

```
tcc -c -Otime -apcs /interwork -DINT_CLOCK dhry_1.c -o dhry_1.o
tcc -c -Otime -DINT_CLOCK dhry_2.c -o dhry_2.o
armcc -c -Otime -apcs /interwork/noswst/nofp time.c -o time.o
armlink -o intdhry dhry_1.o dhry_2.o time.o armlib_i.16l
```

Note that:

- `dhry_1.c` must be compiled for interworking because it contains the routine `main()`, which must be compiled for interworking because an interworking library is being used.
- `time.c` must be compiled for interworking because routines it contains are called from Thumb-compiled code.
- `time.c` is compiled with no software stack checking and no frame pointers so as to be compatible with the Thumb-compiled code.
- An interworking library is used because the application contains ARM–Thumb interworking calls, and a Thumb library is used because `main()` is compiled for Thumb.

For more information see **Chapter 12, Interworking ARM and Thumb**.

- Pull the interrupt routine out from `time.c`, because this is the only routine that has to be compiled for ARM (timer-increment exception handlers are always entered and exited in ARM state). You could then pass `time.c` through `tcc`, and put the interrupt routine into a new file, say `handle.c`, that would be passed through `armcc`. This would remove the need for any ARM/Thumb interworking, as the state change when entering and leaving an exception handler is done automatically.

7.13.3 Results with on-board timer

Below are the results obtained by running the modified Dhrystone applications on each of these particular boards. As can be seen, the addition of the timer has given much more consistent and accurate results.

Runs	PIE7 (Dhrystones)	PIV7T (Dhrystones)
50000	17985.6	14792.9
100000	17985.6	14814.8
200000	17985.6	14825.8

Table 7-3: Modified Dhrystone timings

7.14 Floating-Point and Other Coprocessors

By default, EmbeddedICE assumes that there are no coprocessors attached to the debuggee. If in fact there are coprocessors attached, a suitable `coproc` command should be issued to the debugger.



# 8

## Benchmarking, Performance Analysis and Profiling

8.1	Introduction	8-2
8.2	Measuring Code and Data Size	8-3
8.3	Performance Benchmarking	8-6
8.4	Improving Performance and Code Size	8-14
8.5	Profiling	8-19

# Benchmarking, Performance Analysis and Profiling

---

## 8.1 Introduction

This chapter explains how to run benchmarks on the ARM processor, and shows you how to use the profiling facilities to help improve the size and performance of your code. It makes extensive use of the ARM Software Development Toolkit's example programs, and contains a number of practical exercises for you to follow. You should therefore have access to the toolkit's `examples` directory, and the ARM software tools themselves, while working through it.

When developing application software or comparing the ARM with another processor, it is often useful to measure:

- code and data sizes
- overall execution time
- time spent in specific parts of an application

Such information enables you to:

- compare the ARM's performance against other processors in benchmark tests
- make decisions about the required clock speed and memory configuration of a projected system
- pinpoint where an application can be streamlined, leading to a reduction in the system's memory requirements
- identify performance-critical sections of code which you can then optimize, either by using a more efficient algorithm, or by rewriting in assembler



# Benchmarking, Performance Analysis and Profiling

## 8.2 Measuring Code and Data Size

To measure code size, do not look at the linked image size or object module size, as these include symbolic information that is not part of the binary data. Instead, use one of the following armlink options:

- |                           |   |
|---------------------------|---|
| <code>-info sizes</code>  | which gives a breakdown of the code and data sizes of each object file or library member making up an image           |
| <code>-info totals</code> | which gives a summary of the total code and data sizes of all object files and all library members making up an image |

The information provided by these options can be broken down into:

- code (or read-only) segment
- data (or read-write) segment
- debug data

### Code (or read-only) segment

- |                             |  |
|-----------------------------|--|
| <code>code size</code>      | gives the code size, excluding any data that has been placed in the code segment (see <code>inline data</code> ).  |
| <code>inline data</code>    | reports the size of the read-only data included in the code segment by the compiler.<br><br>Typically, this data contains the addresses of variables that are accessed by the code, plus any floating-point immediate values or immediate values that are too big to load directly into a register. It does not include inlined strings, which are listed separately (see <code>inline strings</code> ). |
| <code>inline strings</code> | shows the size of read-only strings placed in the code segment.<br><br>The compiler puts such strings here whenever possible, because this reduces runtime RAM requirements.   |
| <code>const</code>          | lists the size of any variables explicitly declared as <code>const</code> .<br><br>These variables are guaranteed to be read-only and so are placed in the code segment by the compiler.   |

### Data (or read-write) segment

- |                      |  |
|----------------------|--|
| <code>RW data</code> | gives the size of read-write data.<br><br>This is data that is read-write and also has an initializing value. Read-write data consumes the displayed amount of RAM at runtime, but also requires the same amount of ROM to hold the initializing values that are copied into RAM on image startup. |
|----------------------|--|

# Benchmarking, Performance Analysis and Profiling

---

`0-init data` shows the size of read-write data that is zero-initialized at image startup.

Typically this contains arrays that are not initialized in the C source code. Zero-initialized data requires the displayed amount of RAM at runtime but does not require any space in ROM, since its initializing value is 0.

## Debug data

`debug data` reports the size of any debugging data if the files are compiled with the `-g` option.

**Note** *There are totals for the debug data, even though the code has not been compiled for source-level debugging, because the compiler automatically adds information to an AIF file to allow no frame pointer debugging. See **8.4.1 Compiler options** on page 8-14.*

## 8.2.1 Calculating ROM and RAM requirements

Calculate the ROM and RAM requirements for your system as follows:

```
ROM      Code size + inline data + inline strings + const data
          + RW data

RAM      RW Data + 0-init data
```

In more complex systems, you may require the code segment to be downloaded from ROM into RAM at runtime. Although this increases the system's RAM requirements, this could be necessary if, for example, RAM access times are faster than ROM access times and the execution speed of the system is critical.

## 8.2.2 Code and data sizes example: Dhrystone

The Dhrystone application is located in the `examples` subdirectory of the ARM Software Development Toolkit. Copy the files into your working directory.

### If you are using the command-line tools:

- 1 Compile the Dhrystone files, without linking:

```
armcc -c -DMSC_CLOCK dhry_1.c dhry_2.c
```

The compiler will produce a number of warnings, which you may ignore or suppress using the `-w` option. These are caused by the Dhrystone application being coded in K&R style C rather than ANSI C.

- 2 Perform the link stage, with the `-info totals` option to give a report on the total code and data sizes in the image, broken into separate totals for the object files and library files:

```
armlink -info totals dhry_1.o dhry_2.o armlib.321 -o dhry
```

(If `armlib.321` is not in the current directory, refer to it by its full pathname.)

# Benchmarking, Performance Analysis and Profiling

If you are using the Windows toolkit:

- 1 Load the Dhrystone project file `dhyr.apj` into the *ARM Project Manager (APM)*.
- 2 Change the project setting to produce a release build with a little-endian memory model, using the ARM tools (instead of the Thumb tools —see **2.5.1 Configuring tools** on page 2-20).
- 3 Click the **Force Build** button. This compiles and links the project, automatically generating a summary of the total code and data sizes in the image.



## Results

The results are shown in the following table:

	code size	inline data	inline strings	const data	RW data	zero-init data	debug data
Object totals	2268	28	1448	0	48	10200	64
Library totals	34400	400	736	128	700	1176	416
Grand totals	36668	428	2184	128	748	11376	480

**Table 8-1: Code and data sizes results**

**Note** You may obtain slightly different figures, depending on the version of the compiler, linker and library in use.

# Benchmarking, Performance Analysis and Profiling

---

## 8.3 Performance Benchmarking

### 8.3.1 Measuring performance

There are two debugger internal variables that contain the cycle counts, and these can be displayed using `armsd's print` command, or from **Debugger Internals** from the *ARM Debugger for Windows (ADW)* **View** menu:

<code>\$statistics</code>	shows the total number of each type of cycle since execution started. Refer to an ARM processor datasheet for a explanation of cycle types.
<code>\$statistics_inc</code>	shows the number of cycles of each type since the previous time <code>\$statistics</code> or <code>\$statistics_inc</code> was displayed.
<code>\$statistics_inc_w</code>	outputs the difference between the current statistics and the point at which you asked for the <code>\$statistics_inc_w</code> window.

Make sure you have not compiled with source-level debugging enabled (`armcc -g`), since this causes sub-optimal code to be generated (typically 23–30% bigger and slower). The `-gr` and `-go` compiler options reduce this to 7–15% larger.

If your code makes use of floating-point mathematics, a considerable amount of time may be spent in the *floating-point emulator (FPE)*.

### 8.3.2 Cycle counting example: Dhrystone

In this example, the number of instructions executed by the main loop of the Dhrystone application and the number of cycles consumed are determined. A suitable place to break within the loop is the invocation of function `Proc_5`.

**If you are using the command-line tools:**

- 1 Load the executable, produced in **8.2.2 Code and data sizes example: Dhrystone** on page 8-4, into the debugger:

```
armsd dhry
```

- 2 Set a breakpoint on the first instruction of `Proc_5`:

```
break @Proc_5
```

When prompted, request at least two runs through Dhrystone.

- 3 Once the breakpoint at the start of `Proc_5` has been reached, display the system variable `$statistics` (which gives the total number of instructions and cycles taken so far) and restart execution:

```
print $statistics  
go
```

- 4 When the breakpoint is reached again, you can obtain the number of instructions and cycles consumed by one iteration:

```
print $statistics_inc
```

# Benchmarking, Performance Analysis and Profiling

If you are using the Windows toolkit:



- 1 If you have not already done so, build the Dhrystone project as described in **8.2.2 Code and data sizes example: Dhrystone** on page 8-4.
- 2 Click the **Debug** button on the APM toolbar. ADW is started and the Dhrystone project loaded.
- 3 Locate function `Proc_5` by choosing **Low Level Symbols** from the **View** menu.
- 4 Double-click on `Proc_5` to open the Disassembly Window.
- 5 Toggle the breakpoint on `Proc_5` in the Disassembly Window by selecting the instruction, then clicking the **Toggle breakpoint** button on the toolbar.
- 6 Click the **Go** button to begin execution.  
When prompted, request at least two runs through Dhrystone.
- 7 When the breakpoint set at `main` is reached, click **Go** again to begin execution of the main application.
- 8 Once the breakpoint at `Proc_5` is reached, choose **Debugger Internals** from the **View** menu.
- 9 Double-click on the `statistics_inc` field to display the detail for this variable.
- 10 Click the **Go** button. When the breakpoint at `Proc_5` is reached again, the contents of the `statistics_inc_w` field is updated to reflect the number of instructions and cycles consumed by one iteration of the loop.

## Results

The results are shown in the following table:

Instructions	S-cycles	N-cycles	I-cycles	C-cycles	F-cycles
358	427	188	64	0	0

**Table 8-2: Cycle counting results**

S-cycles	Sequential cycles: the CPU requests transfer to or from the same address, or from an address that is a word or halfword after the preceding address.
N-cycles	Nonsequential cycles: the CPU requests transfer to or from an address that is unrelated to the address used in the preceding cycle.
I-cycles	Internal cycles: the CPU does not require a transfer because it is performing an internal function (or running from cache).
C-cycles	Coprocessor cycles.
F-cycles	Fast clock cycles for cached processors (FCLK).

**Note** You may obtain slightly different figures, depending on the version of the compiler, linker or library in use, and the processor for which the ARMulator is configured.

# Benchmarking, Performance Analysis and Profiling

---

## 8.3.3 Real-time simulation

The ARMulator also provides facilities for real-time simulation. To carry out such a simulation, the ARMulator needs to know:

- the type and speed of the memory attached to the processor
- the speed of the processor

While it is executing your program, the ARMulator counts the total number of clock ticks taken. This allows you to determine how long your application would take to execute on real hardware.

## 8.3.4 Reading the simulated time

When it performs a simulation, the ARMulator keeps track of the total time elapsed. This value may be read either by the simulated program or by the debugger.

### Reading the simulated time from assembler

To read the simulated clock from an assembly language program use SWI 0x61 (SWI\_Clock).

### Reading the simulated time from C

From C, use the standard C library function `clock()`, which returns the number of elapsed centiseconds.

### Reading the simulated time from the debugger

The internal variable `$clock` contains the number of microseconds since simulation started. To display this value, use the command:

```
Print $clock
```

if you are using `armsd`, or choose **Debugger Internals** from the View menu if you are using the ARM Debugger for Windows.

## 8.3.5 Map files

The type and speed of memory in a simulated system is detailed in a *map* file. This defines the number of regions of attached memory, and for each region:

- the address range to which that region is mapped
- the data bus width in bytes
- the access time for the memory region

`armsd` expects the map file to be in the current working directory under the name `armsd.map`.

The ARM Debugger for Windows will accept a map file of any name, provided that it has the extension `.map`. See **8.3.6 Real-time simulation example: Dhrystone** on page 8-12 for details of how to associate a map file into an ADW session.

# Benchmarking, Performance Analysis and Profiling

---

## Format of a Map file

The format of each line is:

```
start size name width access read-times write-times
```

where:

<i>start</i>	is the start address of the memory region in hexadecimal, for example, 80000.
<i>size</i>	is the size of the memory region in hexadecimal, for example, 4000.
<i>name</i>	is a single word that can be used to identify the memory region when the memory access statistics are displayed. This name is of no significance to the debugger, so you can use any name, but to ease readability of the memory access statistics, give a descriptive name such as SRAM, DRAM, EPROM.
<i>width</i>	is the width of the data bus in bytes (that is, 1 for an 8-bit bus, 2 for a 16-bit bus, or 4 for a 32-bit bus).
<i>access</i>	<p>describes the type of access that may be performed on this region of memory: <i>r</i> is for read-only, <i>w</i> for write-only, <i>rw</i> for read-write, or <i>-</i> for no access.</p> <p>An asterisk (*) may be appended to the access to describe a Thumb-based system that uses a 32-bit data bus, but which has a 16-bit latch to latch the upper 16 bits of data, so that a subsequent 16-bit sequential access may be fetched directly out of the latch.</p>
<i>read-times</i>	<p>describes the nonsequential and sequential read times in nanoseconds. These should be entered as the nonsequential read access time followed by / (slash), followed by the sequential read access time. Omitting the / and using only one figure indicates that the nonsequential and sequential access times are the same.</p> <p><b>Note:</b> The times entered should not simply be the speed quoted on top of a memory chip, but should have a 20–30 ns signal propagation time added to them.</p>
<i>write-times</i>	describes the nonsequential and sequential write times. The format is identical to that of read times.

Examples are given below.

# Benchmarking, Performance Analysis and Profiling

---

## Example 1

```
0 80000000 RAM 4 rw 135/85 135/85
```

This describes a system with a single contiguous section of RAM from 0 to 0x7fffffff with a 32-bit data bus, read-write access, and N and S access times of 135ns and 85ns respectively.

This is typical of a 20MHz PIE (Platform Independent Evaluation) board. Note that the N-cycle access time is one clock cycle longer than the S-cycle access time. For a faster system, a smaller N-cycle access time should be used. For example, for a 33MHz system, the access times would be defined as 115/85 115/85.

## Example 2—clock speed 20MHz

```
0 80000000 RAM 1 rw 150/100 150/100
```

This describes a system with the same single contiguous section of memory, but with an 8-bit external data bus and slightly faster access times.

## Example 3—clock speed 20MHz

The following description file details a typical embedded system with 32kb of on-chip memory, 16-bit ROM and 32Kb external DRAM:

```
00000000 8000      SRAM  4 rw 1/1 1/1
00008000 8000      ROM   2 r  100/100 100/100
00010000 8000      DRAM  2 rw 150/100 150/100
7fff8000 8000      Stack 2 rw 150/100 150/100
```

There are four regions of memory:

- A fast region from 0 to 0x7fff with a 32-bit data bus.
- A slower region from 0x8000 to 0xffff with a 16-bit data bus. This is labelled ROM and contains the image code, and is therefore marked as read-only.
- Two sections of RAM, one from 0x10000 to 0x17fff which will be used for image data, and one from 0x7fff8000 to 0x7ffffffffff which will be used for stack data (the stack pointer is initialized to 0x80000000).

Note that in the final hardware, the two distinct regions of the external DRAM would be combined. This does not make any difference to the accuracy of the simulation.

Note that the SRAM region is given access times of 1ns. In effect, this means that each access will take 1 clock cycle, as armsd rounds this up to the nearest clock cycle. However, specifying it as 1ns allows the same map file to be used for a number of simulations with differing clock speeds.

**Note** *To ensure accurate simulations, take care that all areas of memory likely to be accessed by the image you are simulating are described in the memory map.*

To ensure that you have described all areas of memory you think the image should access, you can define a single memory region that covers the entire address range as the map file's last line.



# Benchmarking, Performance Analysis and Profiling

---

For example, to the above description you could add the line:

```
00000000 80000000 Dummy 4 - 1/1 1/1
```

You can then detect if any reads or writes are occurring outside the regions of memory you expect using the `print $memory_statistics` command. This can be a very useful debugging tool.

## Reading the memory statistics

To read the memory statistics use the command:

```
Print $memory_statistics
```

The statistics are reported in the following form

```
address namew accR(N/S)W(N/S)reads(N/S)writes(N/S)time (ns)
```

```
00000000Dummy4 -1/11/1 0/00/00
7FFF8000Stack4 rw135/85135/852852/829829/1456833214
00008000R04 r 70/7070/7012488/3806938069/9025788907
00000000SRAM4 rw135/85135/8527/00/04050
```

`Print $memstats` is a shorthand version of `Print $memory_statistics`.

## Processor clock speed

The debugger also needs details of the clock speed of the processor being simulated. In `armsd`, this is set by the command-line option `-clock value`. The value is presumed to be in Hz unless MHz is specified.

In the ADW, the clock speed is set by configuring the debugger. This is done by selecting **ARMulator** from the **Configure Debugger** sub-menu off the **Options** menu. The value entered in the dialog should be specified in MHz.

# Benchmarking, Performance Analysis and Profiling

---

## 8.3.6 Real-time simulation example: Dhrystone

To work through this example, you need to create a map file. (If one exists in the files you copied from the toolkit directory, edit it to match the one shown here.) Call it `armsd.map`.

```
00000000 80000000 RAM 4 RW 135/85 135/85
```

This describes a system with a single contiguous section of memory 0x80000000 bytes in length, labeled as RAM, starting at address 0x0, with a 32-bit (4-byte) data bus, with both read and write access, and read and write access times of 135ns nonsequential and 80ns sequential.

### If you are using the command-line tools:

- 1 Load the executable produced in **8.2.2 Code and data sizes example: Dhrystone** on page 8-4 into the debugger, telling the debugger that the processor is clocked at 20MHz:

```
armsd -clock 20MHz dhry
```

As the debugger loads, you can see the information about the memory system that the debugger has obtained from the `armsd.map` file.

- 2 Begin execution:  

```
go
```
- 3 When requested for the number of Dhrystones, enter 30000.
- 4 When the application completes, record the number of Dhrystones per second reported. This is your performance figure.

### If you are using the Windows toolkit:

ADW automatically sets to an `armsd.map` file. To change to the map file you have created:

- 1 Choose **Configure Debugger** from the **Options** menu. This displays the Debugger configuration dialog.
- 2 Select the **Debugger** tab to change the default memory map. Click the **Browse** button and select the map file you created.

# Benchmarking, Performance Analysis and Profiling

The association is now set up, and you can run the program.



- 1 Start ADW from APM by clicking the **Debug** icon. If a dialog box prompts you to save the changes to the project file, click **Yes**.
- 2 To set up the debugger to run at the required clock speed:
  - a) Select **Configure Debugger** from the **Options** menu.
  - b) Select **ARMulator** from the **Target Environment** box on Target page of the Debugger Configuration dialog.
  - c) Click the **Configure** button.
  - d) Change the **Clock Speed** to 200MHz and click **OK**.
  - e) Click **OK** on the Debugger Configuration dialog. The application will be reloaded.



- 3 Click the **Go** button to begin execution, and again when the breakpoint on `main` is reached.
- 4 When requested for the number of Dhrystones, enter 30000.
- 5 When the application completes, record the number of Dhrystones per second reported. This is your performance figure.



Once the debugger is configured to emulate a processor of the required clock speed (in this case 20MHz), you can repeat the simulation by clicking on **Execute** rather than **Debug** in APM.

Result: 13452.9 Dhrystones per second

**Note** *You may obtain slightly different figures, depending on the version of the compiler, linker, library in use, and the processor for which the ARMulator is configured.*

## 8.3.7 Reducing the time required for simulation

You may be able to significantly reduce the time taken for a simulation by dividing the specified clock speed by a factor of ten, and multiplying the memory access times by the corresponding factor of ten. Take the time reported by the `clock()` function (or by `SWI_Clock`) and divide by the same factor of ten.

The reason this works is because the simulated time is recorded internally in nanoseconds, but `SWI_Clock` only returns centiseconds. Therefore, dividing the clock speed by ten shifts digits from the nanosecond count into the centisecond count, allowing the same level of accuracy but taking only one tenth of the time to simulate.

# Benchmarking, Performance Analysis and Profiling

---

## 8.4 Improving Performance and Code Size

### 8.4.1 Compiler options

There are two main goals when compiling a benchmark:

- minimizing code size
- maximizing performance

The ARM C compiler has a number of command-line options which control the way in which code is generated. You can find a full list in the C Compiler chapter of the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

By default, the ARM C compiler is highly optimizing. None of the optimizations carried out are dangerous. The code produced from your source is balanced for a compromise of code size versus execution speed. However, there are a number of compiler options that can affect the size and/or the performance of generated code. These may be used individually or may be combined to give the required effect.

`-g` Turns on source-level code debugging. This option severely impacts the size and performance of generated code, since it turns off all compiler optimizations. Use it only when carrying out source-level debugging of your code, and never enable it for a release build.

`-Ospace` Optimizes for code size at the expense of performance.

`-Otime` Optimizes for performance at the expense of size.

Note that `-Ospace` and `-Otime` are complementary. They can be used together on different parts of a build. For example, `-Otime` could be used on time-critical source files, with `-Ospace` being used on the remainder.

`-zpj0` Disables cross-jump optimization. Cross-jump optimization is a space-saving strategy whereby common sections of code at the end of each element in a `switch()` statement are identified and commoned together, each occurrence of the code section being replaced with a branch to the commoned code section. However, this optimization can lead to extra branches being executed which may decrease performance, especially in interpreter-like applications that typically have large `switch()` statements. Use the `-zpj0` option to disable this optimization if you have a time-critical `switch()` statement.

Alternatively, you can use:

```
#pragma nooptimise_crossjump
```

before the function containing the `switch()` and:

```
#pragma optimise_crossjump
```

after it.

# Benchmarking, Performance Analysis and Profiling

---

`-apcs /nofp`

By default, armcc generates code that uses a dedicated frame pointer register. This register holds a pointer to the stack frame and is used by the generated code to access a function's arguments. By specifying `-apcs /nofp` on the command line, you can force armcc to generate code that does not use a frame pointer, but accesses the function's arguments via offsets from the stack pointer.

This means that function entry is simplified (two instructions are saved) and a register is freed up for use as a work register, so your code should be smaller and run more quickly. However, to take full advantage of this, you must use the correct variant of the C library.

**Note:** tcc never uses a frame pointer, so this option does not apply when compiling Thumb code.

`-apcs /noswst`

By default, armcc generates code at the head of each function which checks that the stack has not overflowed. This code can contribute several percent to the code size, so it may be worthwhile disabling this option with `-apcs /noswst`.

Again this means that function entry is simplified, saving a compare and a conditional branch per non-leaf function, and a register is freed up for use as a work register, improving both code size and execution speed. You must use the correct variant of the C library to take full advantage of this.

Be careful to ensure that your program's stack is not going to overflow, or that you have an alternative stack checking mechanism such as an MMU-based check.

**Note:** tcc has stack checking disabled by default.

`-pcc`

The code generated by the compiler can be slightly larger when compiling with the `-pcc` switch. This is because of extra restrictions on the C language in the ANSI standard which the compiler can take advantage of when compiling in ANSI mode.

If your code will compile in ANSI mode, do not use the `-pcc` switch. The Dhrystone application provides a good example. It is written in old-style K&R C, but compiles more efficiently in ANSI mode, even though it causes the compiler to generate a number of warning messages.

`-ARM7T`

This option applies to armcc only. By default, armcc generates code which is suitable for running on processors that implement ARM Architecture 3 (for example, ARM6, ARM7). If you know that the code is going to be run on a processor with halfword support, you can use the `-ARM7T`, `-arch 4` or `-arch 4T` options to instruct the compiler to use the ARM Architecture 4 halfword and signed byte instructions. This can result in significantly improved code density and performance when accessing 16-bit data.

# Benchmarking, Performance Analysis and Profiling

---

## 8.4.2 Improving image size with the linker

You can reduce image size by using the method described in **Chapter 13, Writing code for ROM**, instead of the standard C library, which adds a minimum of around 15KB to an image.

## 8.4.3 Changing the source

You can make further improvements to code size and performance in addition to those achieved by good use of compiler options by modifying the code to take advantage of the ARM processor's features.

### Use of shorts

ARM cores implementing an ARM Architecture earlier than version 4 do not have the ability to directly load or store halfword quantities (or *shorts*). This affects code size. Generally, code generated for Architecture 3 that makes use of shorts will be larger than equivalent code that only performs byte or word transfers. Storing a short is particularly expensive, as the ARM processor must make two byte stores. Similarly, loading a short requires a word load, followed by shifting out the unwanted halfword.

If your core has halfword support, tell the compiler using the `-ARM7T`, `-arch 4` or `-arch 4T` options discussed in **8.4.1 Compiler options** on page 8-14. This ensures that the resulting code contains the Architecture 4 halfword instructions.

If you are writing or porting for cores that do not have halfword support, you should ideally minimize the use of shorts. However, this is sometimes impossible—for instance, when porting C programs from x86 or 68k architectures, which frequently make heavy use of them. If the code has been written with portability in mind, all you may have to do is change a `typedef` or `#define` to use `int` instead of `short`. Where this is not the case, you may have to make some functional changes to the code.

# Benchmarking, Performance Analysis and Profiling

You may be able to establish the extent of code size increase resulting from using shorts by compiling the code with:

```
armcc -Dshort=int
```

which preprocesses all instances of `short` to `int`. Be aware that, although it may compile and link correctly, code created with this option may not function as expected.

Whatever your approach, you will need to weigh the change in code size against the opposite change in data size.

The program below illustrates the effect of using shorts, integers and the `-ARM7T` option on code and data size.

```
typedef short number;
#include <stdio.h>
number array [2000];
number loop;
int main()
{
    for (loop=0; loop < 2000; loop++)
        array[loop] = loop;
    return 0
}
```

The results of compiling the program with all three options are shown in the following table:

	code size	inline data	inline strings	const data	RW data	O-init data	debug data
short	76	8	0	0	4	4000	64
short with hardware support (see note below)	60	8	0	0	4	4000	64
int	44	8	0	0	4	8000	0

**Table 8-3: Object code and data sizes**

**Note** *Hardware support for halfwords selected using `-ARM7T`, `-arch 4` and `arch 4T` options*

## Other changes

- Modify performance-critical C source to compile efficiently on the ARM. See **8.4 Improving Performance and Code Size** on page 8-14.
- Port small, performance-critical routines into ARM assembler.

Use the compiler's `-S` option to produce assembly output, and take this as a starting point for your own hand-optimized assembly language.

# Benchmarking, Performance Analysis and Profiling

---

You can make significant performance improvements by using Load and Store Multiple instructions in memory-intensive algorithms. When optimizing the routines:

- use load/store multiple instructions for memory-intensive algorithms (armcc cannot make full use of LDM/STM due to the difficulty involved in arranging the order of registers).
- use 64-bit result multiply instructions (where available) for fixed-point arithmetic.
- Replace small, performance-critical functions by macros, or use the `__inline` preprocessor directive.
- Avoid the use of `setjmp()` in performance-critical routines (particularly in pcc mode).



# Benchmarking, Performance Analysis and Profiling

---

## 8.5 Profiling

Profiling allows the time spent in specific parts of an application to be examined. It does not require any special compile time or link time options. The only requirement is that low-level symbols must be included in the image. These are inserted by the linker unless it is instructed otherwise by the `-Nodebug` option.

Profiling data is collected by `armsd` or the ADW while the code is being executed. The data is saved to a file, which is then loaded into the ARM profiler which displays the results. The profiler in turn generates a profile report.

### 8.5.1 About `armprof`

The ARM profiler, `armprof`, displays an execution profile of a program from a profile data file generated by either the ADW or by `armsd`. The profiler displays one of two types of execution profile, depending on the amount of information present in the profile data:

- If only PC sampling information is present, the profiler can merely display a flat profile giving the percentage time spent in each function, excluding the time spent in any of its children.
- If function call count information is present, the profiler can display a call graph profile that shows not only the percentage time spent in each function, but also the percentage time accounted for by calls to all children of each function, and the percentage time allocated to calls from different parents.

The compiler automatically prepares the code for profiling, so no special options are required at compile time. At link time, all you have to do is ensure that your program image contains symbols (the linker's default setting).

At this release, you can only profile programs that are loaded into store from the debugger: function call counting for code in ROM is not available (and will not be for the foreseeable future). You must inform the debugger that you wish to gather profile data when the program image is loaded. The debugger then alters the image, diverting calls to counting veneers.

The debuggers allow the collection of PC samples to be turned on and off at arbitrary times, allowing data to be generated only for the part of a program on which attention is focussed (omitting initialization code, for example). However, care should be taken that the time between turning sampling on and off is long compared with the sample interval, or the data generated may be meaningless. Note also that turning sampling on and off does not affect the gathering of call counts.

# Benchmarking, Performance Analysis and Profiling

---

## 8.5.2 Collecting profile data

The debugger collects profiling data while an application is executing. You can turn data collection on and off during execution, so that only the relevant sections of code are profiled:

- If you are using `armsd`, use the `profon` and `profoff` commands.
- If you are using ADW, select **Toggle Profiling** from the **Profiling** menu on the **Options** menu (see **3.4.7 Profiling** on page 3-8).

The format of the execution profile obtained depends on the type of information stored in the data file:

PC sampling	provides a flat profile of the percentage time spent in each function (excluding the time spent in its children)
Function call count	provides a call graph profile showing the percentage time spent in each function, plus the percentage time accounted for by calls to the children of each function, and the percentage time allocated to calls from different parents

The debugger needs to know which profiling method you require when it loads the image. The default is PC sampling. To obtain a call graph profile:

- If you are using `armsd`, load the image with:  
`load/callgraph image-file`
- If you are using the ADW, select **Call Graph Profiling** from the **Profiling** sub-menu on the **Options** menu.

Then execute the code to collect the profile data.

## 8.5.3 Saving profile data

Once collection is complete, save the data to a file:

- If you are using `armsd`, issue the `profwrite` command:  
`profwrite data-file`
- If you are using the ADW, choose **Write to File** from the **Profiling** sub-menu on the **Options** menu.

# Benchmarking, Performance Analysis and Profiling

---

## 8.5.4 Command-line options

A number of options are available to control the format and amount of detail present in the profiler output.

<code>-Parent</code>	Tells the profiler to display information about the parents of each function in the profile listing. This gives information about how much time is spent in each function servicing calls from each of its parents.
<code>-Child</code>	Tells the profiler to display information about the children of each function. The profiler displays the amount of time spent by each child performing services on behalf of the parent.
<code>-NoParent</code>	Turns off the parent listing.
<code>-NoChild</code>	Turns off the child listing.
<code>-Sort Cumulative</code>	Tells the profiler to sort the output by the total time spent in each function and all of its children.
<code>-Sort Self</code>	Tells the profiler to sort the output by the time spent in each function (excluding the time spent in its children).
<code>-Sort Descendants</code>	Tells the profiler to sort the output by the time spent in all of a function's children, but excluding time spent in the function itself.
<code>-Sort Calls</code>	Tells the profiler to sort the output by the number of calls to each function in the listing.

By default, child functions are listed, but not parent functions, and the output is sorted by cumulative time.

### Example

```
armprof -parent sort.prf
```

## 8.5.5 Generating the profile report

The ARM profiler utility, `armprof`, generates the profile report using the data in the file. The report is divided into sections, each of which gives information about a single function in the program.

A section's function (called the *current function*) is indicated by having its name start at the left-hand edge of the `Name` column. If call graph profiling is used, information is also given about child and parent functions. Functions listed below the current function are its children—functions called by it. Those listed above the current function are its parents—functions that call it.

# Benchmarking, Performance Analysis and Profiling

The columns in the report have the following meanings:

Name	Displays the function names. The current function in a section starts at the column's left-hand edge: parent and child functions are shown indented.
cum%	Shows the total percentage time spent in the current function plus the time spent in any functions that it called. It is only valid for the current function.
self%	Shows the percentage time spent in a function. <ul style="list-style-type: none"><li>For the current function, it shows the percentage time spent in this function.</li><li>For parent functions, it shows the percentage time spent in the current function on behalf of the parent.</li><li>For child functions, it shows the percentage time spent in this child on behalf of the current function.</li></ul>
desc%	Shows the percentage time spent in a function. <ul style="list-style-type: none"><li>For the current function, it shows the percentage time spent in children of the current function on the current function's behalf.</li><li>For parent functions, it shows the percentage time spent in children of the current function on behalf of this parent.</li><li>For child functions, it shows the percentage time spent in this child's children on behalf of the current function.</li></ul>
calls	Shows the number of times a function is called. <ul style="list-style-type: none"><li>For the current function, it shows the number of times this function was called.</li><li>For parent functions, it shows the number of times this parent called the current function.</li><li>For child functions, it shows the number of times this child was called by the current function.</li></ul>

Below is a section of the output from armprof for a call graph profile:

Name	cum%	self%	desc%	calls
main	96.04%	0.16%	95.88%	0
qsort		0.44%	0.75%	1
_printf		0.00%	0.00%	3
clock		0.00%	0.00%	6
_sprintf		0.34%	3.56%	1000
check_order		0.29%	5.28%	3
randomise		0.12%	0.69%	1
shell_sort		1.59%	3.43%	1
insert_sort		19.91%	59.44%	1
-----				
main		19.91%	59.44%	1
insert_sort	79.35%	19.91%	59.44%	1
strcmp		59.44%	0.00%	243432
-----				



# Benchmarking, Performance Analysis and Profiling

---

From the `cum%` column, you can see (in the `main` section) that the program spent 96.04 percent of its time in `main` and its children. Of this, only 0.16 percent of the time is spent in `main` (`self%` column), whereas 95.88 percent of the time is spent in functions called by `main` (`desc%` column). The call count for `main` is 0 because it is the top-level function, and is not called by any other functions, whereas the section for `insert_sort` shows that it made 243432 calls to `strcmp`, and that this accounted for 59.44 percent of the time spent in `strcmp` (the `desc%` column shows 0 in this case because `strcmp` does not call any functions).

## 8.5.6 Profiling example: sorts

The `sorts` application can be found in the `Examples` subdirectory of the ARM Software Development Toolkit. Copy the files into your working directory.

### PC sampling information

If you are using the command-line tools:

- 1 Compile the `sorts.c` example program:  

```
armcc -Otime -o sorts sorts.c
```
- 2 Start `armsd` and load the executable:  

```
armsd sorts
```
- 3 Turn profiling on:  

```
profon
```
- 4 Run the program as normal:  

```
go
```
- 5 Once execution completes, write the profile data to a file using the `ProfWrite` command:  

```
ProfWrite sort1.prf
```
- 6 Exit `armsd`:  

```
Quit
```
- 7 The profile for the collected data can now be generated by entering the following at the system prompt:  

```
armprof sort1.prf > prof1
```

The profiler generates report and the output is sent to file `prof1`. This can then be viewed as a text file.

If you are using the Windows toolkit:

- 1 Load the project file `sorts.apj` into APM by choosing **Open** from the **Project** menu.
- 2 Build the project by clicking the **Force Build** button.  
The project is built and any messages are displayed in the build log.



# Benchmarking, Performance Analysis and Profiling

---



- 3 Load the debugger by clicking the **Debug** button.

ADW is started and the application is loaded.

- 4 Turn on profiling in the ARM Debugger for Windows by selecting **Toggle Profiling** from the **Profiling** sub-menu on the **Options** menu.



- 5 Start the program by clicking on **Go** button.

The program runs and stops at the breakpoint on `main`.

- 6 Click the **Go** button again.

The program resumes execution.

- 7 Once execution completes, write the profile data to the file `sort1.prf`, by selecting **Write to file** from the **Profiling** sub-menu on the **Options** menu.

- 8 Exit ADW and start a DOS session. Make the profile directory the current directory. The profile for the collected profile data can now be generated by entering the following at the system prompt:

```
armprof sort1.prf > prof1
```

`armprof` generates the profile report and outputs it to the profile file. This can then be viewed as a text file.

## Call graph information

If you are using the command line tools:

- 1 Restart the debugger:

```
armsd
```

- 2 Load the `sorts` program into `armsd` with the `/callgraph` option:

```
load/callgraph sorts
```

`/callgraph` tells `armsd` to prepare an image for function call count profiling by adding code that counts the number of function calls.

- 3 Turn profiling on:

```
ProfOn
```

- 4 Run the program as normal:

```
go
```

- 5 Once execution completes, write the profile data to a file:

```
ProfWrite sort2.prf
```

- 6 Exit `armsd`:

```
Quit
```

- 7 Generate the profile by entering the following at the system prompt:

```
armprof -Parent sort2.prf > prof2
```

`-Parent` instructs `armprof` to include information about the callers of each function. `armprof` generates the profile report and outputs it to `prof2`, which can then be viewed as a text file.

# Benchmarking, Performance Analysis and Profiling

---

If you are using the Windows toolkit:



1 Reload the debugger by clicking the **Debug** on the APM toolbar.



2 Turn on call graph profiling by selecting **Call graph profiling** from the **Profiling** sub-menu on the **Options** menu.



3 Reload the image by clicking on the **Reload** icon. This forces call graph profiling to take effect.

4 Turn on profiling in ADW by selecting **Toggle Profiling** from the **Profiling** sub-menu on the **Options** menu.

5 Start the program by clicking the **Go** button.

The program runs and stops at the breakpoint on `main`.

6 Click the **Go** button again.

The program resumes execution.

7 Once execution completes, write the profile data to the file `sort2.prf`, by selecting **Write to file** from the **Profiling** sub-menu on the **Options** menu.

8 Exit ADW and invoke a DOS session.

9 Generate the profile by entering the following at the DOS prompt:

```
armprof -Parent sort2.prf > prof2
```

`-Parent` instructs `armprof` to include information about the callers of each function. `armprof` generates the profile report, which is output to `prof2`. This can then be viewed as a text file.

## 8.5.7 Profiling and instruction tracing with ARMulator

In addition to profiling the time spent in specific parts of an application, the ARMulator also provides facilities for profiling other performance statistics, and for generating full instruction traces.

The ARMulator allows:

- enhanced profiling with the Profiler module  
The ARMulator has an Events mechanism, allowing, for example, cache misses, branch mispredictions, etc. to be profiled. The profiling is controlled through a configuration file, rather than from the debugger. However, the data is collected by the debugger and processed by `armprof` in exactly the same way, using the same commands/menus. For example, profiling cache misses allows you to find areas of code that are causing high-levels of cache activity. This code can then be optimized and tuned accordingly.
- instruction tracing with the Tracer module  
At the cost of a significant runtime overhead, the Tracer module can generate a continuous trace stream of executing instructions and memory accesses.

Both modules are supplied in source form, and are user-modifiable. This allows profiling and tracing to be customized to your specific needs. For more details refer to **Chapter 5, The ARMulator**.

# Benchmarking, Performance Analysis and Profiling

---





# 9

## Basic Assembly Language Programming

9.1	Introduction	9-2
9.2	Structure of an ARM Assembler Module	9-5
9.3	Assembler Subroutines	9-7
9.4	Structure of a Thumb Assembler Module	9-8
9.5	Loading Constants into Registers	9-10
9.6	Conditional Execution	9-14
9.7	Loading Addresses into Registers	9-19
9.8	Calling Assembler from C	9-25
9.9	Load and Store Multiple Registers Instructions	9-27

# Basic Assembly Language Programming

---

## 9.1 Introduction

This chapter provides you with a basic, practical understanding of how to write ARM and Thumb assembly language modules. It also gives an insight into some of the facilities provided within the ARM and Thumb Assemblers (armasm and tasm).

It does not provide an in-depth description of either instruction set. This information can be found either in the *ARM Architecture Reference Manual* (ARM DDI 0100), an appropriate ARM data sheet, or on the Quick Reference Card (ARM QRC 0001) provided with this toolkit.

### 9.1.1 Overview of the ARM Architecture

The ARM Architecture has the following key features.

#### 32-bit address space

Processors implementing Versions 1 and 2 of the ARM Architecture only had a 26-bit addressing range. All later ARM processors have a 32-bit addressing range. Those implementing ARM Architectures 3 and 4 (but not 4T) have retained the ability to perform 26-bit addressing for backwards compatibility.

#### Load/store architecture

Only load and store instructions can access memory. This means that:

- data processing operations have to use intermediate registers, loading the data from memory beforehand and storing it back again afterwards. However, this is not as inefficient as one might think. Most operations require several instructions to carry out the required calculation, and each instruction runs as fast as possible instead of being slowed down by external memory accesses.
- there are specific memory access instructions with powerful auto-indexing addressing modes, and the ability to transfer multiple register values in a single instruction.

#### 32-bit and 8-bit data

All ARM cores have load and store instructions that handle data as 32-bit words or 8-bit bytes. Words are always aligned on 4-byte boundaries.

#### 16-bit data

Processors implementing Version 4 or 4T of the ARM Architecture also have load and store instructions for handling 16-bit halfwords. Halfwords are aligned on 2-byte boundaries.

# Basic Assembly Language Programming

---

## 37 registers

These comprise:

- **30 general purpose registers**

Fifteen of these are accessible at any one time as `r0`, `r1`, ..., `r13`, `r14`.

`r13` is usually used as a stack pointer (`sp`).

`r14` is used as a link register to store the return address of subroutine call (`lr`).

- **a Current Program Status Register (CPSR)**

This is used to hold copies of the ALU status flags, the current mode, interrupt disabled bits and, on Thumb-compatible cores, the current state (ARM or Thumb).

- **five Saved Program Status Registers (SPSR)**

These are used to store the CPSR when an exception is taken. One SPSR is accessible in each of the exception-handling modes.

- **a program counter**

This is accessed as `r15` (or `pc`).

The banking of registers gives rapid context switching for dealing with exceptions and privileged operations

## 9.1.2 ARM instruction set summary

### 32-bit instructions

All ARM instructions are 32 bits long, so the core can fetch every instruction from memory in one cycle. In addition, all instructions are stored word-aligned in memory, which means that the bottom two bits of the program counter (`r15`) are always set to zero in ARM state.

### Conditional execution

All instructions are executed conditionally on the value of the ALU status flags in the CPSR. Only data processing operations with the S bit set change the state of these flags.

### Register access

There is no breakdown of the currently accessible registers; all instructions can access `r0`–`r14` and most also allow use of `r15` (`pc`). There are also specific instructions to allow access to the CPSR and SPSRs.

### No single instruction to move an immediate 32-bit value to a register

In general, a literal value must be loaded from memory. However, a large set of common 32-bit values can be generated in a single instruction.

# Basic Assembly Language Programming

---

## Inline barrel shifter

The second argument to all ARM data-processing and single data-transfer operations can be shifted in quite a general way before the operation is performed. This supports, but is not limited to, scaled addressing, multiplication by a constant, and the construction of constants, within a single instruction.

## Coprocessor instructions

These support a general way to extend the ARM's Architecture in a customer-specific manner.

### 9.1.3 Thumb instruction set summary

#### 16-bit instructions

All Thumb instructions are 16 bits long, so the core can fetch every instruction from memory in one cycle. In addition, all instructions are stored halfword-aligned in memory, which means that the bottom bit of the program counter (`r15`) is always set zero in Thumb state.

In effect, the Thumb instruction set is actually a subset of the full ARM instruction set that has been compressed into 16-bit opcodes. This subset is one that is highly optimized for production by a compiler.

#### Register access

The accessible registers are broken down into two groups:

- Lo Registers  
`r0–r7` : general purpose registers that are fully accessible.
- Hi Registers  
`r8–r12` : limited accessibility registers that can be used, for instance, as fast temporary storage. Only certain instructions can access these.  
`r13 (sp)`, `r14 (lr)` and `r15 (pc)` : limited accessibility registers, with certain instructions having implicit access to these.

Only indirect access to the CPSR is allowed (most data processing instructions automatically update the ALU status flags in Thumb), and no access at all is allowed to the SPSRs.

#### No single instruction to move an immediate 32-bit value to a register

A literal value must be loaded from memory, unless it lies in the range 0–255. However, a large set of common 32-bit values can be generated in a single instruction.

#### Conditional execution

Only branch instructions may be conditionally executed, depending on the value of the ALU status flags in the CPSR. Most data processing operations cause the status flags to be updated.

# Basic Assembly Language Programming

## 9.2 Structure of an ARM Assembler Module

The following simple example that illustrates some of the core constituents of an ARM assembler module. This file can be found as `armex.s` in the `examples\asm` subdirectory of the toolkit.

```
AREA ARMex, CODE, READONLY; Name this block of code.
ENTRY                      ; Mark first instruction
                           ; to execute.
start MOV    r0, #10        ; Set up parameters
      MOV    r1, #3
      ADD    r0, r0, r1     ; r0 = r0 + r1
stop  MOV    r0, #0x18      ; angel_SWIreason_ReportException
      LDR    r1, =0x20026   ; ADP_Stopped_ApplicationExit
      SWI    0x123456       ; Angel semihosting ARM SWI.
      END                      ; Mark end of file.
```

To build the module at the command line, type the command:

```
armasm -g armex.s
```

The object code can then be linked to produce an executable:

```
armlink armex.o -o armex
```

This can then be loaded into `armsd` and executed:

```
armsd armex
```

### 9.2.1 Description of the module

#### The AREA directive

Areas are independent chunks of data or code that are manipulated by the linker. A complete application consists of one or more areas. The linker places each area in the application image according to the area placement rules: see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for more information on the linker. This means that areas adjacent in source modules are not necessarily be adjacent in the application image.

In the source file, the start of an area is marked by the AREA directive, which names the area and sets its attributes. The attributes are placed after the name, separated by commas. The syntax is of the directive is:

```
AREA    name {, attribute} {, attribute} ...
```

Any name can be used, but certain choices are conventional: for example, `|C$$code|` is used for code areas produced by the C compiler, or for code areas otherwise associated with the C library. The areas attributes can be placed in any order. If the AREA directive is missing, the assembler will generate an AREA with an unlikely name (`|$$$$$$|`).

# Basic Assembly Language Programming

---

This example consists of a single area that contains CODE and is marked as being READONLY. A single code area is the minimum required to produce an application, though typically a DATA area that is marked as being READWRITE is available.

## The ENTRY directive

The first instruction to be executed within an application is marked by the ENTRY directive. Because an application cannot have more than one entry point, the ENTRY directive can appear in only one of the modules. Note that when an application contains some C code, the entry point is often contained within the C library.

## General layout

The general form of source lines in an assembler module is:

```
label instruction ; comment
```

Note that the three sections are separated by at least one whitespace character (such as a space or a tab). Instructions never start in the first column, since they must be preceded by white space even if there is no label. All three sections are optional, and the assembler also accepts blank lines to improve the clarity of the code.

Source lines may be up to 255 characters long. A single source statement can be continued over several lines by placing the backslash character (\) at the end of each line. The backslash must be followed immediately by the end-of-line character. This sequence is treated by the assembler as white space. It must not be used within quoted strings.

## General code description

The application code begins executing at `start`, where it loads the decimal values 10 and 3 into registers `r0` and `r1`. These registers are then added together and the result placed in `r0`.

## Application termination

After executing the main code, the application terminates by returning control back to the debugger. This is done using the Angel semihosting SWI (by default this is `0x123456` in ARM state), with the following parameters:

- `r0` equal to `angel_SWIreason_ReportException` (by default `0x18`)
- `r1` equal to `ADP_Stopped_ApplicationExit` (by default `0x20026`)

For further information on this, see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

## The END directive

This directive causes the assembler to stop processing this source file. Every assembly language source module must therefore finish with it.

# Basic Assembly Language Programming

## 9.3 Assembler Subroutines

Subroutines can be implemented in assembler using a `BL label` instruction to call the subroutine and a `MOV pc,lr` instruction at the end of the subroutine to return.

Parameters can be passed using registers (typically `r0-r3`) and results returned similarly. The example below illustrates this. This file can be found as `subrout.s` in the `examples\asm` subdirectory of the toolkit.

```
        AREA subrout, CODE, READONLY ; Name this block of code.
        ENTRY                        ; Mark first instruction
                                           ; to execute.

start   MOV    r0, #10                ; Set up parameters.
        MOV    r1, #3
        BL     doadd                 ; Call subroutine

stop    MOV    r0, #0x18              ; angel_SWIreason_ReportException
        LDR    r1, =0x20026          ; ADP_Stopped_ApplicationExit
        SWI    0x123456              ; Angel semihosting ARM SWI.

doadd   ADD     r0, r0, r1            ; Subroutine code.
        MOV     pc, lr               ; Return from subroutine.
        END                                     ; Mark end of file
```

To build the module at the command line, type the command:

```
armasm -g subrout.s
```

The object code can then be linked to produce an executable:

```
armlink subrout.o -o subrout
```

This can then be loaded into `armsd` and executed:

```
armsd subrout
```

# Basic Assembly Language Programming

---

## 9.4 Structure of a Thumb Assembler Module

The following is a simple example of a Thumb assembler module, based on `subrout.s`. This file can be found as `thumbsub.s` in the `examples\asm` subdirectory of the toolkit.

```
AREA ThumbSub, CODE, READONLY; Name this block of code.
ENTRY                          ; Mark first instruction to
                               ; execute.

CODE32                         ; Subsequent instructions are
                               ; ARM.
                               ; Header

ADR r0,start + 1; Processor starts in ARM
                               ; state,
BX      r0                    ; so small ARM code header used
                               ; to call Thumb main program.

CODE16                         ; Subsequent instructions are
                               ; Thumb.

start
    MOV r0, #10                ; Set up parameters.
    MOV r1, #3
    BL doadd                    ; Call subroutine
stop  MOV r0, #0x18             ; angel_SWIreason_ReportException
    LDR r1, =0x20026; ADP_Stopped_ApplicationExit
    SWI 0xAB                    ; Angel semihosting Thumb SWI
doadd ADD r0, r0, r1            ; Subroutine code
    MOV pc, lr                  ; Return from subroutine.
END                             ; Mark end of file.
```

To build the module at the command line, type the command:

```
tasm -g thumbsub.s
```

The object code can then be linked to produce an executable:

```
armlink thumbsub.o -o thumbsub
```

This can then be loaded into `armsd` and executed:

```
armsd thumbsub
```

### Description of the module

The Thumb assembler (`tasm`) is capable of assembling both Thumb and ARM code. By default, it assumes that the module contains Thumb code. However, because the processor always starts up in ARM state, the first few lines required in a standalone Thumb assembler module will always be ARM code, which will carry out the switch into Thumb state. This ARM code is prefixed by the directive `CODE32`. The main body of the module is then prefixed by



# Basic Assembly Language Programming

---

CODE16 as it is Thumb code. Following the CODE16 directive, the action of the main code is identical to that of the ARM example given in session 1, as these particular instructions are identical in both instruction sets.

Note that:

- The ADR generates the address by loading r0 with the value “PC+offset”. This will be examined further in this chapter.
- Full details of mixing ARM and Thumb instructions in this fashion are given in ***Chapter 12, Interworking ARM and Thumb.***
- The Thumb semihosting SWI is, by default, a different number to the ARM semihosting SWI (0xab rather than 0x123456).

# Basic Assembly Language Programming

## 9.5 Loading Constants into Registers

### 9.5.1 Why is loading constants an issue?

Because all ARM instructions are precisely 32 bits long, and do not use the instruction stream as data, there is no single instruction that will load a 32-bit immediate constant into a register without performing a data load from memory.

The same applies to Thumb instructions which are 16 bits long.

Although a data load places any 32-bit value in a register, there are more direct—and therefore more efficient—ways to load many commonly-used constants.

### 9.5.2 Direct loading with MOV/MVN in ARM state

The MOV instruction allows 8-bit constant values to be loaded directly into a register, giving a range of 0x0 to 0xff (255). The bitwise complement of these values can be constructed using MVN, giving the added ability to load values in the range 0xfffffff0 to 0xffffffff.

Further constant values can be generated by using MOV and MVN in conjunction with the barrel shifter. These further constants are 8-bit values rotated right through an even number of positions (giving rotate rights of 0, 2, 4...28, 30). Thus MOV can directly load values that follow the pattern shown below:

Decimal values	Equivalent hexadecimal	Step between values	Rotate information
0–255	0–0xff	1	No rotate
256, 260, 264, ..., 1020	0x100–0x3fc	4	Right by 30 bits
1024, 1040, 1056, ..., 4080	0x400–0xff0	16	Right by 28 bits
4096, 4160, 4224, ..., 16320	0x1000–0x3fc0	64	Right by 26 bits
etc	etc	etc	etc

Table 9-1: MOV load values

MVN can then load the bitwise complements of these values.

#### Using of MOV and MVN

It is possible to use MOV and MVN by specifying the 8-bit constant, followed by the rotate right value. For instance:

```
MOV r0, #0xFF,30      ; r0 = 1020
```



# Basic Assembly Language Programming

However, converting a constant into this form is an onerous task. Therefore, when given a constant, the assembler attempts to carry out the conversion itself. If the supplied constant cannot be expressed as a rotated 8-bit value or its bitwise complement, the assembler reports this as an error. So, the above example would more typically be written as:

```
MOV r0, #0x3FC      ; r0 = 1020
```

thus allowing the assembler to do the conversion. The instructions shown in **Table 9-2: User instructions** illustrate how this works. The left-hand column lists the ARM instructions entered by the user, while the right-hand column shows the assembler's attempts to convert the supplied constants to an acceptable form.

User instruction	Assembled equivalent
MOV r0, #0	MOV r0, #0
MOV r1, #0xFF000000	MOV r1, #0xFF, 8
MOV r2, #0xFFFFFFFF	MVN r2, #0
MVN r3, #1	MVN r3, #1
MOV r4, #0xFC000003	MOV r4, #0xFF, 6
MOV r5, #0x03FFFFFFC	MVN r5, #0xFF, 6
MOV r6, #0x55555555	Error (cannot be constructed)

Table 9-2: User instructions

### 9.5.3 Direct loading with MOV in Thumb state

In Thumb state only constants, in the range 0–255 can be directly loaded. This is because:

- The Thumb MOV instruction does not provide inline access to the barrel shifter, and therefore constants cannot be right-rotated in the manner of the ARM-state move instruction.
- The Thumb MVN instruction can act only on registers and not on constant values. Thus bitwise complements cannot be directly loaded as they can in ARM state.

If the user attempts to use a MOV with a value outside this range, the assembler generates an error.

### 9.5.4 Direct loading with LDR Rd, =const

The ARM and Thumb assemblers both provide a mechanism that, unlike MOV (and MVN in ARM state), can construct any 32-bit numeric constant in a single instruction, but which may not result in a data processing operation to do it. This is the LDR Rd, =const pseudo-instruction.



# Basic Assembly Language Programming

---

If the constant specified in an `LDR Rd, =const` pseudo-instruction can be constructed with a `MOV` or `MVN`, the assembler generates the appropriate instruction. Otherwise it produces an `LDR` instruction with a PC-relative address to read the constant from a literal pool (a portion of memory embedded in the code to hold constant values).

By default, a literal pool is placed at every `END` directive. However, for large modules, this may not be accessible throughout the program:

- In ARM state, the offset from the PC to the constant must be less than 4KB.
- In Thumb state, the offset from the PC to the constant must be less than 1KB (and greater than 8 bytes).

If this default literal pool is out of range, further pools can be placed in code by adding an `LTORG` directive (usually between subroutines so that the processor never tries to execute the constants as instructions).

When an `LDR Rd, =const` pseudo-instruction needs to access a constant in a literal pool, the assembler first checks previously encountered literal pools to see whether the desired constant is already available and addressable. If so, it addresses the existing constant, otherwise it attempts to place the constant in the next available literal pool. If this is not addressable, because it does not exist or is too far away, an error results. In such cases, an additional `LTORG` should be placed close to (but after) the failed `LDR Rd, =const` pseudo-instruction.

To see how this works in practice, consider the following example, which is available as `loadcon.s` in the `examples\asm` subdirectory of the toolkit. The instructions listed as comments are the ARM instructions which are generated by the assembler:

```
AREA Loadcon, CODE, READONLY
ENTRY                               ; Mark first instruction.
start BL func1                      ; Branch to first subroutine.
      BL func2                      ; Branch to second subroutine.
stop  MOV r0, #0x18                 ; angel_SWIreason_ReportException
      LDR r1, =0x20026              ; ADP_Stopped_ApplicationExit
      SWI 0x123456                 ; Angel semihosting ARM SWI
func1
      LDR r0, =42                   ; => MOV R0, #42
      LDR r1, =0x55555555          ; => LDR R1, [PC, #offset to
                                   ; Literal Pool 1]
      LDR r2, =0xFFFFFFFF          ; => MVN R2, #0
      MOV pc, lr
      LTORG                        ; Literal Pool 1 contains
                                   ; literal &55555555.
```

# Basic Assembly Language Programming

---

```
func2
    LDR    r3, =0x55555555    ; => LDR R3, [PC, #offset to
                                ; Literal Pool 1]
    ; LDR r4, =0x66666666    ; If this is uncommented it
                                ; will fail, as Literal Pool 2
                                ; is not accessible (out of reach).

    MOV    pc, lr
LargeTable%    4200            ; Clears a 4200 byte area of memory,
                                ; starting at the current location,
                                ; to zero.

    END                        ; Literal Pool 2 is empty.
```

**Note**    *The literal pools are placed outside sections of executed code. This typically means placing them between subroutines, as is done here, if more pools than the default one at END is required.*

To build the module at the command line, type the command:

```
armasm -g loadcon.s
```

The object code can then be linked to produce an executable:

```
armlink loadcon.o -o loadcon
```

This can then be loaded into armsd and executed:

```
armsd loadcon
```

# Basic Assembly Language Programming

## 9.6 Conditional Execution

### 9.6.1 The ARM's ALU status flags

The ARM's Current Program Status Register contains, among other flags, copies of the ALU status flags:

- N Negative result from ALU flag
- Z Zero result from ALU flag
- C ALU operation Carried out
- V ALU operation oVerflowed

Data processing instructions change the state of the ALU's N, Z, C and V status outputs, but these are latched in the CPSR's ALU flags only if a special bit (the S bit) is set in the instruction.

### 9.6.2 ARM state execution conditions

Every ARM instruction has a 4-bit field that encodes the conditions under which it will be executed. These conditions refer to the state of the ALU N, Z, C and V flags as shown in **Table 9-3: Field mnemonics**.

Field mnemonic	Condition
EQ	Z set (equal)
NE	Z clear (not equal)
CS/HS	C set (unsigned >=)
CC/LO	C clear (unsigned <)
MI	N set (negative)
PL	N clear (positive or zero)
VS	V set (overflow)
VC	V clear (no overflow)
HI	C set and Z clear (unsigned >)
LS	C clear and Z set (unsigned <=)
GE	N and V the same (signed >=)

Table 9-3: Field mnemonics



# Basic Assembly Language Programming

Field mnemonic	Condition
LT	N and V differ (signed <)
GT	Z clear, N and V the same (signed >)
LE	Z set, N and V differ (signed <=)
AL	Always execute (the default if none is specified)

**Table 9-3: Field mnemonics (Continued)**

If the condition field indicates that a particular instruction should not be executed given the current settings of the status flag, the instruction simply soaks up one cycle but has no other effect.

If the current instruction is a data processing instruction, and the flags are to be updated by it, the instruction must be suffixed by an S. The exceptions to this are CMP, CMN, TST and TEQ, which always update the flags (since this is their only effect).

Examples:

```
ADD    r0, r1, r2    ; r0 = r1 + r2, don't update flags.
ADDS   r0, r1, r2    ; r0 = r1 + r2 and update flags.
ADDEQS r0, r1, r2    ; If Z flag set then r0 = r1 + r2,
                    ; and update flags.
CMP     r0, r1        ; update flags based on r0-r1.
```

## 9.6.3 Using conditional execution in ARM state

Consider Euclid's Greatest Common Divisor algorithm:

```
function gcd (integer a, integer b) : result is integer
while (a <> b) do
    if (a > b) then
        a = a - b
    else
        b = b - a
    endif
endwhile
result = a
```

# Basic Assembly Language Programming

---

This might be coded with only conditional execution of branches as:

```
gcd
    CMP    r0, r1
    BEQ    end
    BLT    less
    SUB    r0, r0, r1
    B      gcd
less
    SUB    r1, r1, r0
    B      gcd
end
```

Using this code, every time a branch is taken three cycles will be wasted in refilling the pipeline and continuing execution from the new location. (The other instructions and non-executed branches will take up a single cycle.)

**Note** *On ARM8 and StrongARM, this pipeline refill period may be shorter because of the existence of branch prediction hardware on these ARMs.*

Also, because of the number of branches in the code, the code is seven instructions long. Using the conditional execution feature of the ARM instruction set, the algorithm can be recoded to improve both its execution time and code density:

```
gcd
    CMP    r0, r1
    SUBGT  r0, r0, r1
    SUBLT  r1, r1, r0
    BNE    gcd
```

Not only has code size been reduced from seven words to four, but execution time has also decreased, as can be seen by comparing **Table 9-4: Conditional branches only** on page 9-17 and **Table 9-5: All instructions conditional** on page 9-17. These show the execution times for the simple case where r0 equals 1 and r1 equals 2. In this case, replacing branches with conditional execution of all instructions has given a saving of three cycles. With all inputs to the gcd algorithm, the conditional version of the code will execute in the same number of cycles (when both inputs are the same), or fewer cycles.

## 9.6.4 Thumb state execution conditions

Unlike the ARM instruction set, not all instructions in the Thumb instruction set can be conditionally executed; the only instruction that can be is the conditional branch instruction.

In addition, in Thumb state most instructions automatically set the S bit and so always cause the ALU status flags in the CPSR to be updated following data processing instructions.

The only data processing instructions that do not do this are the MOV and ADD instructions when a Hi registers is involved.



# Basic Assembly Language Programming

r0:a	r1: b	Instruction	Cycles
1	2	CMP r0, r1	1
1	2	BEQ end	Not executed - 1
1	2	BLT less	3
1	2	SUB r1, r1, r0	1
1	2	BAL gcd	3
1	1	CMP r0, r1	1
1	1	BEQ end	3
			Total = 13

**Table 9-4: Conditional branches only**

r0:a	r1: b	Instruction	Cycles <sup>1</sup>
1	2	CMP r0, r1	1
1	2	SUBGT r0, r0, r1	Not executed -1
1	1	SUBLT r1, r1, r0	1
1	1	BNE gcd	3
1	1	CMP r0, r1	1
1	1	SUBGT r0, r0, r1	Not executed -1
1	1	SUBLT r1, r1, r0	Not executed -1
1	1	BNE gcd	Not executed -1
			Total = 10

**Table 9-5: All instructions conditional**

# Basic Assembly Language Programming

---

The gcd algorithm would have to be coded in a similar fashion to its original ARM form (where only branches were conditional). Consequently, it now occupies seven instructions, but because the instructions are only 16-bits long rather than 32-bits long, the overall code size is only 14 bytes, rather than the 16 bytes that the second ARM implementation occupied.

In addition, it would actually run more quickly than the second ARM implementation would if 16-bit memory were being used (which would typically be the case in Thumb-based systems), as only one memory access would be required for each Thumb instruction, whereas each ARM instruction would require two fetches.

# Basic Assembly Language Programming

## 9.7 Loading Addresses into Registers

It is often necessary to load a register with an address, for example the location of a string constant within the code segment, or the start location of a jump table. However, because ARM code is inherently relocatable, and because there are limitations on the values that can be directly moved into a register, absolute addressing cannot be used for this purpose. Instead, addresses must be expressed as offsets from the current PC. A register can either be directly set by combining the current PC with the appropriate offset, or the address can be loaded from a literal pool.

### 9.7.1 LDR Rd, =label

As well as numeric constants, the `LDR Rd, =` pseudo-instruction can cope with PC-relative expressions (labels).

Instructions such as:

```
LDR    r0, =source
```

cause the address of the label `source` to be stored in a nearby literal pool and a PC-relative LDR instruction to be generated, which stays appropriate wherever the AREA containing the LDR and the literal pool are located by the linker. Note that if an appropriate literal already exists in an accessible literal pool, the assembler uses it rather than creating a new instance.

The following example illustrates how this works. The instruction listed in the comment is the ARM instruction that is generated by the assembler. This file can be found as `ldrlabel.s` in the `examples\asm` subdirectory of the toolkit.

```
AREA LDRLabel, CODE, READONLY
ENTRY                               ; Mark first instruction.

start
    BL    func1                    ; Branch to first subroutine.
    BL    func2                    ; Branch to second subroutine.
stop  MOV    r0, #0x18              ; angel_SWIreason_ReportException
    LDR    r1, =0x20026             ; ADP_Stopped_ApplicationExit
    SWI    0x123456                ; Angel semihosting ARM SWI

func1
    LDR    r0, =start              ; => LDR R0,[PC, #offset to
                                   ; Litpool 1]
    LDR    r1, =Darea + 12         ; => LDR R1,[PC, #offset to
                                   ; Litpool 1]
    LDR    r2, =Darea + 6000      ; => LDR R2, [PC, #offset to
                                   ; Litpool 1]
    MOV    pc, lr                 ; Return
    LTORG                          ; Literal Pool 1
```

# Basic Assembly Language Programming

---

```
func2
    LDR    r3, =Darea + 6000 ; => LDR r3, [PC, #offset to
                                ; Litpool 1]
                                ; (sharing with previous literal).
    ; LDR r4, =Darea + 6004 ; If uncommented will produce an
                                ; error as Litpool 2 is out of range.
    MOV    pc, lr            ; Return
Darea    % 8000              ; Clears a 8000 byte area of memory,
                                ; starting at the current location,
                                ; to zero.
    END                    ; Literal Pool 2 is out of range of
                                ; the LDR instructions above.
```

As well as enabling addresses to be generated to labels within the same area, this mechanism can also be used to allow the address of labels in different areas to be generated. This is because the literal value placed in the literal pool (ie. the address of the label) is resolved at link time, but the PC-relative LDR remains constant.

To build the module at the command line, type the command:

```
armasm -g ldrlabel.s
```

The object code can then be linked to produce an executable:

```
armlink ldrlabel.o -o ldrlabel
```

This can then be loaded into armsd and executed:

```
armsd ldrlabel
```

## 9.7.2 An LDR Rd, =label example: string copying

The following is a simple ARM code example that copies one string over the top of another string. This file can be found as `strcpy.s` in the `examples\asm` subdirectory of the toolkit.

```
AREA    StrCopy, CODE, READONLY
ENTRY                                ; Mark the first instruction to call.
start   LDR    r1, =srcstr           ; Pointer to first string
        LDR    r0, =dststr          ; Pointer to second string
        BL     strcpy               ; Call subroutine to do copy.
stop    MOV     r0, #0x18             ; angel_SWIreason_ReportException
        LDR    r1, =0x20026          ; ADP_Stopped_ApplicationExit
        SWI    0x123456              ; Angel semihosting ARM SWI
strcpy
        LDRB   r2, [r1],#1           ; Load byte and update address.
        STRB   r2, [r0],#1           ; Store byte and update address.
```

# Basic Assembly Language Programming

---

```
CMP    r2, #0           ; Check for zero terminator.
BNE    strcpy           ; Keep going if not.
MOV    pc,lr            ; Return
```

```
AREA   Strings, DATA, READWRITE
srcstr DCB "First string - source",0
dststr DCB "Second string - destination",0
END
```

DCB or “Define Constant Byte” is an assembler directive to allocate one or more bytes in memory. It is therefore a useful way to create a string in an assembly language module.

Notice the use of the post-indexed addressing mode to update the address registers in the LDR and STR instructions. For example:

```
LDRB   r2, [r1], #1
```

This causes contents of the address pointed to by `r1` to be loaded into `r2`. After this access is made, `r1` is incremented by 1.

To build the module at the command line, type the command:

```
armasm -g strcpy.s
```

The object code can then be linked to produce an executable:

```
armlink strcpy.o -o strcpy
```

This can then be loaded into `armsd` and executed:

```
armsd strcpy
```

If this module were converted into Thumb code, post-indexed addressing would not be available and so the LDR/STR and address register updates would have to be carried out separately using an ADD instruction following the LDR/STR. For example:

```
LDRB   r2, [r1]
ADD    r1, #1
```

# Basic Assembly Language Programming

## 9.7.3 The ADR and ADRL pseudo-instructions

Sometimes it is important that loading an address does not perform a memory access. The assembler provides two pseudo-instructions, ADR and ADRL, which make it easier to do this. ADR and ADRL accept a PC-relative expression (a label) and calculate the offset required to reach that location.

ADR attempts to produce a single instruction (either an ADD or a SUB) to load an address into a register. If the desired address cannot be constructed in a single instruction, an error is raised. In typical usage, the offset range is 255 bytes for an offset to a non word-aligned address, and 1020 bytes (255 words) for an offset to a word-aligned address.

Note that:

- The label used with ADR or ADRL must be within the same code area. There is no guarantee that the label will be within range after linking if it resides in another area.
- In Thumb state, ADR can only generate word-aligned addresses.

ADRL attempts to produce two data-processing instructions to load an address into a register. Even if it is possible to produce a single data processing instruction to load the address, a second, redundant instruction is produced (this is a consequence of the strict two-pass nature of the assembler). In cases where it is not possible to construct the address using two data-processing instructions, ADRL produces an error, and in such cases the LDR, =label pseudo-instruction is probably the best alternative. In typical usage, the range of an ADRL is 64KBytes for a non-word aligned address and 256KBytes for a word-aligned address.

**Note** *ADRL is not available when assembling Thumb instructions. It is only understood by tasm within sections of ARM code.*

The following example shows how this works. The instructions listed in the comments are the ARM instructions generated by the assembler. This file can be found as adrlabel.s in the examples\asm subdirectory of the toolkit.

```
        AREA adrlabel, CODE,READONLY
        ENTRY                               ; Mark first instruction.

Start
        BL func                             ; Branch to subroutine.

stop    MOV    r0, #0x18                    ; angel_SWIreason_ReportException
        LDR     r1, =0x20026                ; ADP_Stopped_ApplicationExit
        SWI     0x123456                    ; Angel semihosting ARM SWI
        LTORG                                ; Create a literal pool.

func    ADR     r0, Start                    ; => SUB r0, PC, #offset to Start
        ADR     r1, DataArea                ; => ADD r1, PC, #offset to DataArea
        ; ADR r2, DataArea+4300            ; This would fail as the offset
                                           ; cannot be expressed by operand2
                                           ; of an ADD.
```

# Basic Assembly Language Programming

```
        ADRL r3, DataArea+4300    ; => ADD r2, PC, #offset1
                                   ;    ADD r2, r2, #offset2
        MOV    pc, lr             ; Return
DataArea%    8000                ; Clears a 8000 byte area of memory,
                                   ; starting at the current location,
                                   ; to zero.

END
```

To build the module at the command line issue the command:

```
armasm -g adrlabel.s
```

The object code can then be linked to produce an executable:

```
armlink adrlabel.o -o adrlabel
```

This can then be loaded into armsd and executed:

```
armsd adrlabel
```

## 9.7.4 An ADR Rd, label example: jump table

The following is a simple ARM code example that implements a jump table. This file can be found as `jump.s` in the `examples\asm` subdirectory of the toolkit.

```
AREA Jump, CODE, READONLY ; Name this block of code.
num EQU2                  ; Number of entries in jump table.
ENTRY                     ; Mark the first instruction to call.
start MOV r0, #0           ; Set up the three parameters.
      MOV r1, #3
      MOV r2, #2
      BL arithfunc         ; Call the function.
stop  MOV r0, #0x18        ; angel_SWIreason_ReportException
      LDR r1, =0x20026     ; ADP_Stopped_ApplicationExit
      SWI 0x123456        ; Angel semihosting ARM SWI

arithfunc                 ; Label the function.
      CMP r0, #num         ; Treat function code as unsigned
                                   ; integer.
      BHS DoAdd            ; If code is >=2 then do operation 0.
      ADR r3, JumpTable    ; Load address of jump table.
      LDR pc, [r3,r0,LSL#2]; Jump to the appropriate routine.
JumpTable
      DCD DoAdd
      DCD DoSub
```

# Basic Assembly Language Programming

---

```
DoAdd ADD    r0, r1, r2      ; Operation 0, >1
      MOV    pc, lr          ; Return
DoSub SUB    r0, r1, r2      ; Operation 1
      MOV    pc,lr           ; Return
      END                      ; Mark the end of this file.
```

The function `arithfunc()` takes three arguments. The first controls the operation carried out on the second and third arguments. The result of the operation is passed back to the caller routine in `r0`. The operations of the function are:

0 : Result = argument2 + argument3

1 : Result = argument2 - argument3

Values outside this range have the same effect as value 0.

`EQU` is an assembler directive that is used to give a value to a label name. In this example it assigns `num` the value 2. Thus when `num` is used elsewhere in the code, the value 2 is substituted (similar to using a `#define` to set up a constant in C).

`DCD` declares one or more words. In this case each `DCD` stores a single word; the address of a routine to handle a particular clause of the jump table. This can then be used to implement the jump using:

```
LDR pc, [r3,r0,LSL#2]
```

The `LDR` instruction loads the address of the required clause of the jump table into the PC. This is done by multiplying the clause number by 4 (to give a word offset), adding this to the address of the jump table, and then loading the contents of the combined address into the PC (from the appropriate `DCD`).

To build the module at the command line, type the command:

```
armasm -g jump.s
```

The object code can then be linked to produce an executable:

```
armlink jump.o -o jump
```

This can then be loaded into `armsd` and executed:

```
armsd strcpy
```

If this module were to be converted into Thumb code, the `LDR` instruction used to implement the jump would have to be modified because in Thumb state, `LDR` and `STR` cannot increment their base registers. In addition, `LDR` cannot load a value into the PC, nor can it do an inline shift of a value held in a register.

Thus the equivalent code to cause the jump to the appropriate routine would be:

```
LSL    r0, r0,#2
LDR    r3, [r3,r0]
MOV    pc, r3
```

An `ALIGN` directive would also need to be placed before the `JumpTable` label, to ensure that the table is aligned on a 32-bit boundary.



# Basic Assembly Language Programming

## 9.8 Calling Assembler from C

ARM has defined a function interface standard called the *ARM Procedure Call Standard (APCS)*. This defines how independent routines pass data between each other. Under the APCS, the first four arguments to a function are passed in registers `r0` to `r3` (any further parameters being passed on the stack) and a single word result is returned in `r0`. Using this standard, it is therefore possible to mix calls between C and assembler routines. A similar standard, the *Thumb Procedure Call Standard (TPCS)*, exists for Thumb code. For more information on the APCS and TPCS, see **Chapter 10, Using the Procedure Call Standards**.

The following is a simple C program that copies one string over the top of another string, using a call to an assembler subroutine. This file can be found as `strtest.c` in the `examples\asm` subdirectory of the toolkit.

```
#include <stdio.h>
extern void strcpy(char *d, char *s);
int main()
{
    char *srcstr = "First string - source ";
    char *dststr = "Second string - destination ";

    printf("Before copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    strcpy(dststr,srcstr);
    printf("After copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    return (0);
}
```

The following ARM assembler module implements the string copy subroutine. This file can be found as `scopy.s` in the `examples\asm` subdirectory of the toolkit.

```
AREA  SCopy, CODE, READONLY
EXPORT strcpy

strcpy
    ; r0 points to destination string.
    ; r1 points to source string.
    LDRB  r2, [r1],#1; Load byte and update address.
    STRB  r2, [r0],#1; Store byte and update address.
    CMP   r2, #0     ; Check for zero terminator.
    BNE   strcpy     ; Keep going if not.
    MOV   pc,lr      ; Return.
END
```

# Basic Assembly Language Programming

---

To build the assembler module at the command line, type the command:

```
armasm -g scopy.s
```

To build the C module, type the command:

```
armcc -c -g strtest.c
```

The object code can then be linked to produce an executable:

```
armlink strtest.o scopy.o path armlib.32l -o strtest
```

where *path* gives the location of the ARM library, which will normally be the `lib` subdirectory of the toolkit.

This can then be loaded into `armsd` and executed:

```
armsd strtest
```

# Basic Assembly Language Programming

## 9.9 Load and Store Multiple Registers Instructions

### 9.9.1 Multiple versus single transfers

Multiple register transfer instructions provide an efficient way of moving the contents of several registers to and from memory. The advantages of using a multiple register transfer instruction instead of a series of single data transfer instructions are:

- smaller code size
- there is only a single instruction fetch overhead, rather than many instruction fetches
- only one register writeback cycle is required for a multiple register load, as opposed to one for every single register load
- on uncached ARM processors, the first word of data transferred by a load or store multiple is always a nonsequential memory cycle, but all subsequent words transferred can be sequential (faster) memory cycles

### 9.9.2 ARM state instructions

#### The register list

The registers transferred by the LDM and STM instructions are encoded into the instruction by one bit for each of the registers `r0` to `r15`. A set bit indicates that the register will be transferred, and a clear bit indicates that it will not be transferred. Thus it is possible to transfer any subset of the registers in a single instruction.

The subset of registers to be transferred is specified by listing them in curly brackets. For example:

```
{r1, r4-r6, r8, r10}
```

#### Increment/decrement, before/after

The base address for the transfer can either be incremented or decremented between register transfers, and this can happen either before or after each register transfer:

```
STMIA r10, {r1, r3-r5, r8}
```

The suffix `IA` could also have been `IB`, `DA` or `DB`, where `I` indicates increment, `D` decrement, `A` after and `B` before.

**Note** *In all cases, the lowest numbered register is transferred to or from the lowest memory address, and the highest numbered register to or from the highest address. The order in which the registers appear in the register list makes no difference.*

The ARM always performs sequential memory accesses in increasing memory address order. Therefore, decrementing transfers perform a subtraction first, and then increment the transfer address register by register.

# Basic Assembly Language Programming

---

## Base register writeback

Unless specifically requested, the base register is not updated at the end of a multiple register transfer instruction. To specify register writeback, you must use the `!` character:

```
LDMDB r11!, {r9, r4-r7}
```

## Stack notation

Since the load and store multiple instructions have the facility to update the base register (which for stack operations can be the stack pointer), these instructions provide single instruction push and pop operations for any number of registers (LDM being pop, and STM being push).

The Load and Store Multiple Instructions can be used with several types of stack:

- ascending or descending

A stack is able to grow upwards, starting from a low address and progressing to a higher address (an ascending stack), or downwards, starting from a high address and progressing to a lower one (a descending stack).

- empty or full

The stack pointer can either point to the top item in the stack (a full stack), or the next free space on the stack (an empty stack).

As stated above, pop and push operations for these stacks can be implemented directly by load and store multiple instructions. To make it easier for the programmer, special stack suffixes can be added to the LDM and STM instructions (as an alternative to Increment/Decrement and Before/After suffixes) as follows:

```
STMFA r13!, {r0-r5}; Push onto a Full Ascending Stack.
LDMFA r13!, {r0-r5}; Pop from a Full Ascending Stack.
STMFD r13!, {r0-r5}; Push onto a Full Descending Stack.
LDMFD r13!, {r0-r5}; Pop from a Full Descending Stack.
STMEA r13!, {r0-r5}; Push onto an Empty Ascending Stack.
LDMEA r13!, {r0-r5}; Pop from an Empty Ascending Stack.
STMED r13!, {r0-r5}; Push onto Empty Descending Stack.
LDMED r13!, {r0-r5}; Pop from an Empty Descending Stack.
```

Note the use of `r13` as the base pointer here. By convention `r13` is used as the system stack pointer (`sp`). In addition, the system stack is usually Full Descending. This is always the case if the C compiler is being used.

These stack operations are very useful at subroutine entry and exit. At the start of a subroutine, any working registers required can be stored on the stack, and at exit they can be pulled off again. In addition, if the LR is also pushed onto the stack at entry, further subroutine calls can then safely be made, without causing the return address to be lost. The subroutine return can then be made by simply popping the PC off the stack at exit (rather than popping LR and then moving that value into the PC).

# Basic Assembly Language Programming

For example:

```
subroutine
    STMFD sp!, {r5-r7,lr}; Push work registers and LR
    :
    BL somewhere_else
    :
    LDMFD sp!, {r5-r7,pc}; Pop work registers and PC
```

## Block copy example

The following is a simple ARM code example that copies a set of words from a source location to a destination by copying a single word at a time. This file can be found as `word.s` in the `examples\asm` subdirectory of the toolkit.

```
        AREA Word, CODE, READONLY; Name this block of code.
num      EQU      20                ; Set number of words to be copied.
        ENTRY      ; Mark the first instruction to
                        ; call.

start    LDR      r0, =src          ; r0 = pointer to source block
        LDR      r1, =dst          ; r1 = pointer to destination block
        MOV      r2, #num          ; r2 = number of words to copy

wordcopy
        LDR      r3, [r0], #4      ; load a word from the source and
        STR      r3, [r1], #4      ; store it to the destination.
        SUBS     r2, r2, #1        ; Decrement the counter.
        BNE      wordcopy          ; ... copy more.
stop     MOV      r0, #0x18         ; angel_SWIreason_ReportException
        LDR      r1, =0x20026      ; ADP_Stopped_ApplicationExit
        SWI      0x123456          ; Angel semihosting ARM SWI

        AREA BlockData, DATA, READWRITE
src       DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst       DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        END
```

To build the assembler module at the command line, type the command:

```
armasm -g word.s
```

The object code can then be linked to produce an executable:

```
armlink word.o -o word
```

This can then be loaded into `armsd` and executed:

```
armsd word
```

# Basic Assembly Language Programming

---

This module can be made more efficient by using LDM and STM for as much of the copying as possible. A sensible number of words to transfer in one go, given the number of registers that the ARM has, is eight. The number of eight-word multiples in the block to be copied can be found (if `r2` = number of words to be copied) using:

```
MOVS r3, r2, LSR #3; number of eight word multiples
```

This value can then be used to control the number of iterations through a loop that copies eight words per iteration. When there are less than eight words left, the number of words left can be found (presuming that `r2` has not been corrupted) using:

```
ANDS r2, r2, #7; number of words left to copy
```

This value is then used to control the number of iterations through a loop that copies one word per iteration.

Thus `word.s` can be modified to create a new version of the module, which can be found as `block.s` in the `examples\asm` subdirectory of the toolkit.

```
        AREA Block, CODE, READONLY    ; Name this block of code.
num     EQU     20                      ; Set number of words to be
                                         ; copied.

        ENTRY                          ; Mark the first instruction to
                                         ; call.

start   LDR     r0, =src                ; r0 = pointer to source block
        LDR     r1, =dst                ; r1 = pointer to destination
                                         ; block.

        MOV     r2, #num                ; r2 = number of words to copy.
        MOV     sp, #0x400              ; Set up user_mode stack pointer
                                         ; (r13).

blockcopy
        MOVS    r3, r2, LSR #3          ; Number of eight word
                                         ; multiples.

        BEQ     copywords               ; Less than eight words to move?
        STMFD   sp!, {r4-r11}           ; Save some working registers.

octcopy
        LDMIA   r0!, {r4-r11}           ; Load 8 words from the source
        STMIA   r1!, {r4-r11}           ; and put them at the
                                         ; destination.

        SUBS    r3, r3, #1              ; Decrement the counter.
        BNE     octcopy                 ; ... copy more.
        LDMFD   sp!, {r4-r11}           ; Don't need these now-restore
                                         ; originals.
```

# Basic Assembly Language Programming

```
copywords
    ANDS    r2, r2, #7           ; Number of odd words to copy.
    BEQ     stop                ; No words left to copy?

wordcopy
    LDR     r3, [r0], #4         ; load a word from the source and
    STR     r3, [r1], #4         ; store it to the destination.
    SUBS    r2, r2, #1           ; Decrement the counter.
    BNE     wordcopy            ; ... copy more.
stop    MOV     r0, #0x18        ; angel_SWIreason_ReportException
    LDR     r1, =0x20026         ; ADP_Stopped_ApplicationExit
    SWI     0x123456             ; Angel semihosting ARM SWI
    AREA    BlockData, DATA, READWRITE
src     DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst     DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    END
```

This module also uses LDMs and STMs for stack operations, storing the eight registers used for the block copy onto the stack before copying starts, and restoring them when it has finished.

To build the module at the command line, type the command:

```
armasm -g block.s
```

The object code can then be linked to produce an executable:

```
armlink block.o -o block
```

This can then be loaded into armsd and executed:

```
armsd block
```

## 9.9.3 Thumb-state instructions

The Thumb instruction set contains two pairs of multiple register transfer instructions: LDM/STM for block memory transfers and PUSH/POP for stack operations.

### LDM and STM

These instructions can be used to load or store any subset of the 16 registers from or to memory (for block copies). The base address for the transfer is always incremented after each register transfer. Thus the only valid suffix for these instructions is `IA`. In addition, the base register is always updated at the end of the multiple register transfer instruction, meaning that the `!` character must always be specified. Thus, examples of these instructions are:

```
LDMIA r1!, {r0,r2-r7}
STMIA r4!, {r0-r3}
```

# Basic Assembly Language Programming

---

**Note**     *The lowest numbered register is transferred to or from the lowest memory address accessed, and the highest numbered register to or from the highest address accessed. The order in which the registers appear in the register list makes no difference.*

## **PUSH and POP**

These instructions can be used to push any subset of the lo-registers and (optionally) the LR onto the stack, and to pop any subset of the lo registers and (optionally) the PC off the stack (the base address of which will be held in `r13`). Examples of these instructions are:

```
PUSH {r0-r3}
POP {r0-r3}
PUSH {r4-r7,lr}
POP {r4-r7,pc}
```

The optional addition of the LR/PC to the register list is to provide support for subroutine entry and exit.

## **Thumb-state block copy example**

The block copy example `block.s` can be converted into Thumb instructions. An example conversion can be found as `tblock.s` in the `examples\asm` subdirectory of the toolkit.

Note that because the Thumb LDM/STM instructions can access only the lo registers, the number of words copied per iteration has been cut from 8 down to 4. In addition the LDM/STM instructions can be used to carry out the single word at a time copy, as they update the base pointer after each access. If LDR/STR were used for this, separate ADD instructions would be required to update each base pointer.

```
        AREA Tblock, CODE, READONLY; Name this block of code.
num      EQU 20                      ; Set number of words to be copied.
        ENTRY                      ; Mark first instruction to
                                   ; execute.
        CODE32                     ; Subsequent instructions are ARM.
header
        MOV sp, #0x400             ; Set up user_mode stack pointer
                                   ; (r13).
ADR      r0, start + 1              ; Processor starts in ARM state,
        BX r0                      ; so small ARM code header used
                                   ; to call Thumb main program.
        CODE16                     ; Subsequent instructions are
                                   ; Thumb.
start
        LDR r0, =src                ; r0 =pointer to source block
```



# Basic Assembly Language Programming

```
LDR r1, =dst                ; r1 =pointer to destination block
MOV r2, #num                ; r2 =number of words to copy

blockcopy
    LSR r3,r2, #2            ; Number of four word multiples.
    BEQ copywords           ; Less than four words to move?
    PUSH {r4-r7}            ; Save some working registers.

quadcopy
    LDMIA r0!, {r4-r7}      ; Load 4 words from the source
    STMIA r1!, {r4-r7}      ; and put them at the destination.
    SUB r3, #1              ; Decrement the counter.
    BNE quadcopy           ; ... copy more.
    POP {r4-r7}            ; Don't need these now-restore
                           ; originals.

copywords
    MOV r3, #3              ; Bottom two bits represent number
    AND r2, r3              ; ...of odd words left to copy.
    BEQ stop                ; No words left to copy?

wordcopy
    LDMIA r0!, {r3}         ; load a word from the source and
    STMIA r1!, {r3}         ; store it to the destination.
    SUB r2, #1              ; Decrement the counter.
    BNE wordcopy           ; ... copy more.

stop    MOV r0, #0x18        ; angel_SWIreason_ReportException
        LDR r1, =0x20026     ; ADP_Stopped_ApplicationExit
        SWI 0xAB            ; Angel semihosting Thumb SWI

AREA BlockData, DATA, READWRITE
src    DCD 1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst    DCD 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
END
```

To build the module at the command line, type the command:

```
tasm -g tblock.s
```

The object code can then be linked to produce an executable:

```
armlink tblock.o -o block
```

This can then be loaded into armsd and executed:

```
armsd tblock
```



# 10

## Using the Procedure Call Standards

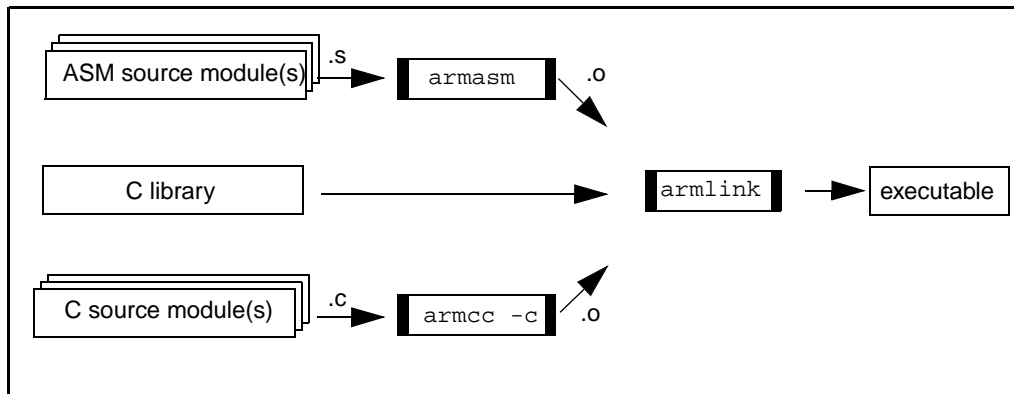
10.1	Introduction	10-2
10.2	Using APCS	10-3
10.3	Using the Thumb Procedure Call Standard	10-9
10.4	Passing and Returning Structures	10-11

# Using the Procedure Call Standards

## 10.1 Introduction

Sometimes you will find it necessary to combine C and assembly language in the same program. For example, performance-critical routines may have to be hand-coded to run at optimum speed.

The ARM Software Development Toolkit allows you to generate *ARM Object Format (AOF)* files from C and assembly language source, and then to link them with one or more libraries to produce an executable file, as shown in **Figure 10-1: Mixing C and assembly language**:



**Figure 10-1: Mixing C and assembly language**

Irrespective of the language in which they are written, routines that make calls to other modules need to observe a common convention of argument and result passing. For the ARM processor, this convention is known as the *ARM Procedure Call Standard (APCS)*. This chapter introduces the APCS, and discusses its role in ARM assembly language for passing and returning values and pointers to structures for use by C routines.

For the full specification of the APCS, see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

## 10.2 Using APCS

APCS is a set of rules governing calls between functions in separately compiled or assembled code fragments.

The APCS defines:

- constraints on the use of registers
- stack conventions
- the format of a stack backtrace data structure
- argument passing and result return
- support for the ARM shared library mechanism

Code produced by compilers is expected to adhere to the APCS at all times; such code is said to be *strictly conforming*. Handwritten code is expected to adhere to the APCS only when making calls to externally visible functions; such code is said to be *conforming*.

The APCS comprises a family of variants. Each is exclusive, so code that conforms to one cannot be used with code that conforms to another. Your choice of variant depends on whether:

- the program counter is 32-bit or 26-bit
- stack limit checking is explicit (performed by code) or implicit (performed by memory management hardware)
- floating-point values are passed in floating-point registers
- code is re-entrant or non re-entrant

### 10.2.1 APCS register names and usage

**Table 10-1: APCS registers** on page 10-4 summarizes the names and uses allocated to the ARM and floating-point registers under the APCS. (Note that not all ARM systems support floating-point. Refer to **Chapter 15, Floating-Point Support** for further details.)

# Using the Procedure Call Standards

Register	APCS name	APCS role
r0	a1	argument 1 / integer result / scratch register
r1	a2	argument 2 / scratch register
r2	a3	argument 3 / scratch register
r3	a4	argument 4 / scratch register
r4	v1	register variable
r5	v2	register variable
r6	v3	register variable
r7	v4	register variable
r8	v5	register variable
r9	sb/v6	static base / register variable
r10	sl/v7	stack limit / stack chunk handle / register variable
r11	fp	frame pointer
r12	ip	scratch register / new-sb in inter-link-unit calls
r13	sp	lower end of current stack frame
r14	lr	link address / scratch register
r15	pc	program counter
f0	0	FP argument 1 / FP result / FP scratch register
f1	1	FP argument 2 / FP scratch register
f2	2	FP argument 3 / FP scratch register
f3	3	FP argument 4 / FP scratch register

Table 10-1: APCS registers

To summarize:

- a1-a4, f0-f3are used to pass arguments to functions. a1 is also used to return integer results, and f0 to return FP results. These registers can be corrupted by a called function.
- v1-v5, f4-f7are used as register variables. They must be preserved by called functions.
- sb, sl, fp, ip, sp, lr, pchave a dedicated role in some APCS variants, although certain registers may be used for other purposes even when strictly conforming to the APCS. In some variants of the APCS sb and sl are available as additional variable registers v6 and v7 respectively. (For more details, see **10.2.3 A more detailed look at APCS register usage** on page 10-8.)



# Using the Procedure Call Standards

As stated previously, hand-coded assembler routines do not need to *conform strictly* to the APCS, but need only *conform*. This means that any register not used in its APCS role by an assembler routine (eg. `fp`) can be used as a working register, provided that its value on entry is restored before returning.

## 10.2.2 An example of APCS register usage: 64-bit integer addition

This example illustrates how to use ARM assembly language to code a small function so that it can be used by C modules.

The function performs a 64-bit integer addition using a two-word data structure to store each 64-bit operand. We will consider the following stages:

- writing the function in C
- examining the compiler's output
- modifying the compiler's output
- looking at the effects of the APCS
- revisiting the first implementation

### Writing the function in C

In assembler, the obvious way to code the addition of double-length integers is to use the Carry flag from the low word addition in the high word addition. However, in C there is no way of specifying the Carry flag, so you have to use a workaround, as follows:

```
void add_64(int64 *dest, int64 *src1, int64 *src2)
{ unsigned hibit1=src1->lo >> 31, hibit2=src2->lo >> 31, hibit3;
  dest->lo=src1->lo + src2->lo;
  hibit3=dest->lo >> 31;
  dest->hi=src1->hi + src2->hi +
    ((hibit1 & hibit2) || (hibit1!= hibit3));
  return;
}
```

The highest bits of the low words in the two operands are calculated (shifting them into bit 0, while clearing the rest of the register). These bits are then used to determine the value of the carry bit (in the same way as the ARM itself does).

### Examining the compiler's output

If the addition routine were to be used a great deal, an implementation such as this would probably be inadequate. To consider the quality of the implementation, examine the code produced by the compiler:

- 1 Copy file `examples/candasm/add64_1.c` (which contains the above C code) to your current working directory.
- 2 Compile it to ARM assembly language source as follows:  

```
armcc -li -apcs 3/32bit -S add64_1.c
```

# Using the Procedure Call Standards

---

The `-s` flag tells the compiler to produce ARM assembly language source (suitable for `armasm`) instead of object code. The `-li` flag tells it to compile for little-endian memory and the `-apcs` option specifies the 32-bit version of APCS 3. You can omit these options if your compiler is configured to have them as defaults.

The output in file `add64_1.s` reveals that this is an inefficient implementation (instructions may vary between compiler releases):

```
add_64
    STMDB    sp!, {v1,lr}
    LDR      v1,[a2,#0]
    MOV      a4,v1,LSR #31
    LDR      ip,[a3,#0]
    MOV      lr,ip,LSR #31
    ADD      ip,v1,ip
    STR      ip,[a1,#0]
    MOV      ip,ip,LSR #31
    LDR      a2,[a2,#4]
    LDR      a3,[a3,#4]
    ADD      a2,a2,a3
    TST      a4,lr
    TEQEQ    a4,ip
    MOVNE    a3,#1
    MOVEQ    a3,#0
    ADD      a2,a2,a3
    STR      a2,[a1,#4]!
    LDMIA    sp!, {v1,pc}
```

## Modifying the compiler's output

Because the Carry flag cannot be specified in C, you need to get the compiler to produce almost the right code, and then modify it by hand. Start with (incorrect) code that does not perform the carry addition:

```
void add_64(int64 *dest, int64 *src1, int64 *src2)
{ dest->lo=src1->lo + src2->lo;
  dest->hi=src1->hi + src2->hi;
  return;
}
```

- 1 Copy file `examples/candasm/add64_2.c` (which contains the above C code) to your current working directory.
- 2 Compile it to ARM assembly language source as follows:

```
armcc -li -apcs 3/32bit -S add64_2.c
```

You can omit the `-li` flag and the `-apcs` option if your compiler is configured to have them as defaults.

You can find the source produced in the file `add64_2.s`. (The code may vary slightly from that shown below as output depends on the version of `armcc` you are using.)



# Using the Procedure Call Standards

---

```
add_64
    LDR    a4,[a2,#0]
    LDR    ip,[a3,#0]
    ADD    a4,a4,ip
    STR    a4,[a1,#0]
    LDR    a2,[a2,#4]
    LDR    a3,[a3,#4]
    ADD    a2,a2,a3
    STR    a2,[a1,#4]
    MOV    pc,lr
```

Comparing this to the C source, you can see that the first `ADD` instruction produces the low order word, and the second produces the high order word. All you need to do to correct this is to get the carry from the low to high word by changing:

- the first `ADD` to `ADDS` (add and set flags)
- the second `ADD` to an `ADC` (add with carry)

You can find this modified code in the directory `examples/candasm` as `add64_3.s`.

## Looking at the effects of the APCS

The most obvious effect of the APCS on the above code is the change in register names:

- `a1` holds a pointer to the destination structure.
- `a2` and `a3` hold pointers to the operand structures.
- `a4` and `ip` are used as temporary registers which are not preserved. (The conditions under which `ip` can be corrupted are discussed in **10.2.3 A more detailed look at APCS register usage**.)

This is a simple leaf function that uses few temporary registers, so none are saved to the stack and restored on exit. Therefore you can use a simple `MOV pc,lr` to return.

If you wish to return a result—perhaps the carry out from the addition—you need to load it into `a1` prior to exit. You can do this as follows:

- 1 Change the second `ADD` to `ADCS` (add with carry and set flags).
- 2 Add the following instructions to load `a1` with 1 or 0 depending on the carry out from the high order addition.

```
    MOV    a1, #0
    ADC    a1, a1, #0
```

## Revisiting the first implementation

Although the first C implementation is inefficient, it shows more about the APCS than the hand-modified version.

# Using the Procedure Call Standards

---

You have already seen `a4` and `ip` being used as non-preserved temporary registers. However, here `v1` and `lr` are also used as temporary registers; `v1` is preserved by being stored (together with `lr`) on entry. Register `lr` is corrupted, but a copy is saved onto the stack and reloaded into `pc` when `v1` is restored. This means that there is still only a single exit instruction, but now it is:

```
LDMIA    sp!, {v1, pc}
```

## 10.2.3 A more detailed look at APCS register usage

Although `sb`, `s1`, `fp`, `ip`, `sp` and `lr` are dedicated registers, the example in **10.2.2 An example of APCS register usage: 64-bit integer addition** shows `ip` and `lr` being used as temporary registers. Indeed, there are times when these registers are not used for their APCS roles. The details given below will enable you to write efficient (but safe) code, that uses as many of the registers as possible, and avoids unnecessary saving and restoring of registers:

<code>ip</code>	is used only during function calls. It is conventionally used as a local code generation temporary register. At other times it can be used as a corruptible temporary register.
<code>lr</code>	holds the address to which control must return on function exit. It can be (and often is) used as a temporary register after pushing its contents onto the stack. This value can then be reloaded straight into the program counter.
<code>sp</code>	is the stack pointer; it is always valid in <i>strictly conforming</i> code, but need only be preserved in handwritten code. Note, however, that if any handwritten code makes use of the stack, <code>sp</code> must be available.
<code>s1</code>	is the stack limit register. If stack limit checking is explicit (ie. is performed by code when stack pushes occur, rather than by memory management hardware causing a trap on stack overflow), <code>s1</code> must be valid whenever <code>sp</code> is valid. If stack checking is implicit, <code>s1</code> is also called as <code>v7</code> , an additional register variable (which must be preserved by called functions).
<code>fp</code>	is the frame pointer register, if the variant of APCS uses <code>fp</code> . This register contains either zero, or a pointer to the most recently created stack backtrace data structure. As with the stack pointer, the frame pointer must be preserved, but in handwritten code it does not need to be available at every instant. However, it must be valid whenever any <i>strictly conforming</i> function is called.
<code>sb</code>	is the static base register. If the variant of the APCS in use is re-entrant, this register is used to access an array of static data pointers to allow code to access data re-entrantly. However, if the variant being used is not re-entrant, <code>sb</code> is instead available as an additional register variable, <code>v6</code> (which must be preserved by called functions).

`sp` must be preserved on function exit for APCS *conforming* code. Register `s1`, `fp` and `sb` must be preserved in the APCS variants that define a use for these registers.

# Using the Procedure Call Standards

## 10.3 Using the Thumb Procedure Call Standard

The *Thumb Procedure Call Standard (TPCS)* is a set of rules that govern inter-calling between functions written in Thumb code. The TPCS is essentially a cut-down APCS.

There are fewer options with TPCS than with the APCS; this reflects the different ways in which ARM and Thumb code are used, and also reflects the reduced nature of the Thumb instruction set.

Specifically, the TPCS does not support:

- disjoint stack extension (stack chunks)  
under the TPCS, the stack must be contiguous. However, this does not prohibit the use of multiple stacks to implement co-routines, for example
- re-entrancy (calling the same entry point with different sets of static data)  
re-entrancy can still be implemented at a user level, by placing in a struct all variables that need to be multiply instantiated, and passing each function a pointer to the struct
- direct floating-point support  
Thumb cannot have access to floating-point (FP) instructions without switching to ARM state. Floating-point is supported indirectly by defining how FP values are passed to and returned from Thumb functions in the Thumb registers

For the full specification of the TPCS, refer to the chapter describing the Thumb Procedure Call Standard in the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

### 10.3.1 TPCS register names and usage

The Thumb register subset has:

- eight visible general-purpose registers ( $r0-r7$ )
- a stack pointer (SP)
- a link register (LR)
- a program counter (PC)

In addition, the Thumb subset can access the other ARM registers ( $r8-r12$ ) singly using a set of special instructions; see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for details.

In the context of the TPCS, each Thumb register has a special name and function as shown in **Table 10-2: TPCS registers**.

# Using the Procedure Call Standards

Register	TPCS name	TPCS role
0	a1	argument 1 / scratch register / FP result / integer result
1	a2	argument 2 / scratch register / FP result
2	a3	argument 3 / scratch register / FP result
3	a4	argument 4 / scratch register
4	v1	register variable
5	v2	register variable
6	v3	register variable
7	v4/wr	register variable/work register in function entry/exit
8	(v5)	(ARM v5 register; no defined role in Thumb)
9	(v6)	(ARM v6 register; no defined role in Thumb)
10	sl	stack limit (no defined role in Thumb)
11	fp	frame pointer (no defined role in Thumb)
12	(ip)	(ARM ip register; no defined role in Thumb) (May be used as a temporary register on Thumb function entry/exit)
13	sp	stack pointer (full descending)
14	lr	link address
15	pc	program counter

Table 10-2: TPCS registers



## 10.4 Passing and Returning Structures

This section covers:

- the default method for passing structures to and from functions
- cases in which passing structures is automatically optimized
- telling the compiler to return a struct value in several registers

### 10.4.1 The default method

Unless special conditions apply (as detailed in following sections), C structures are passed in registers which, if necessary, overflow onto the stack and are returned via a pointer to the memory location of the result.

For struct-valued functions, a pointer to the location where the struct result is to be placed is passed in `a1` (the first argument register). The first argument is then passed in `a2`, the second in `a3`, and so on. It is as if:

```
struct s f(int x)
```

were compiled as:

```
void f(struct s *result, int x)
```

Consider the following code:

```
typedef struct two_ch_struct
{
    char ch1;
    char ch2;
}
two_ch;

two_ch max( two_ch a, two_ch b )
{
    return (a.ch1 > b.ch1) ? a : b;
}
```

This code is available in the file `examples/candasm/two_ch.c`, and can be compiled to produce assembly language source using:

```
armcc -S two_ch.c -li -apcs 3/32bit
```

where `-li` and `-apcs 3/32bit` can be omitted if `armcc` has been configured with these as defaults.

Here is the code `armcc` produces (the version of `armcc` supplied with your release may produce slightly different output to that listed here):

```
max
    MOV     ip,sp
    STMDB   sp!,{a1-a3,fp,ip,lr,pc}
    SUB     fp,ip,#4
    LDRB    a3,[fp,#-&14]
```

# Using the Procedure Call Standards

---

```
LDRB    a2,[fp,#-&10]
CMP      a3,a2
SUBLE    a2,fp,#&10
SUBGT    a2,fp,#&14
LDR      a2,[a2,#0]
STR      a2,[a1,#0]
LDMDB    fp,{fp,sp,pc}
```

The `STMDB` instruction saves the arguments onto the stack, together with the frame pointer, stack pointer, link register and current program counter value (this sequence of values is the stack backtrace data structure).

As one might expect, `a2` and `a3` are then used as temporary registers to hold the required part of the structures passed, and `a1` is a pointer to an area in memory in which the resulting struct is placed.

## 10.4.2 Returning integer-like structures

The APCS specifies different rules for returning *integer-like* structures. An integer-like structure:

- is no larger than one word in size
- has addressable subfields, all of which have an offset of 0

The following structures are integer-like:

```
struct
{
    unsigned a:8, b:8, c:8, d:8;
}

union polymorphic_ptr
{
    struct A *a;
    struct B *b;
    int      *i;
}
```

whereas the structure used in the previous section's example is not:

```
struct { char ch1, ch2; }
```

An integer-like structure has its *contents* returned in `a1`. This means that `a1` is not needed to pass a pointer to a result struct in memory, and is instead used to pass the first argument.

For example, consider the following code:

```
typedef struct half_words_struct
{ unsigned field1:16;
  unsigned field2:16;
} half_words;

half_words max( half_words a, half_words b )
```

# Using the Procedure Call Standards

---

```
{ half_words x;  
  x = (a.field1 > b.field1) ? a : b;  
  return x;  
}
```

Arguments `a` and `b` will be passed in registers `a1` and `a2`, and since `half_word_struct` is integer-like, you would expect `a1` to return the result structure directly, rather than a pointer to it.

This code is available in the file `examples/candasm/half_str.c`, and can be compiled to produce assembly language source using:

```
armcc -S half_str.c -li -apcs 3/32bit
```

where `-li` and `-apcs 3/32bit` can be omitted if `armcc` has been configured with these as defaults.

Here is the code `armcc` produces (the version of `armcc` supplied with your release may produce slightly different output to that listed here):

```
max  
      MOV     a3,a1,LSL #16  
      MOV     a3,a3,LSR #16  
      MOV     a4,a2,LSL #16  
      MOV     a4,a4,LSR #16  
      CMP     a3,a4  
      MOVLW   a1,a2  
      MOV     pc,lr
```

From this you can see that the contents of the `half_words` structure is returned directly in `a1` as expected.

## 10.4.3 Returning non integer-like structures in registers

There are occasions when a function needs to return more than one value. The usual way to achieve this is to define a structure that holds all the values to be returned, and to pass a pointer to the structure back in `a1`. The pointer is then dereferenced, allowing the values to be stored.

For applications in which such a function is time-critical, the overhead involved in “wrapping” and then “unwrapping” the structure can be significant. In this case, you can tell the compiler that a structure should be returned in the argument registers `a1 – a4`, by using the keyword `__value_in_regs`.

Clearly this is only useful for returning structures that are no larger than four words.

# Using the Procedure Call Standards

## example: returning a 64-bit result

To illustrate how to use `__value_in_regs`, consider a function which multiplies two 32-bit integers together and returns a 64-bit result.

To make such a function work, you need to split the two 32-bit numbers (`a`, `b`) into high and low 16-bit parts (`a_hi`, `a_lo`, `b_hi`, `b_lo`). You then need to perform the four multiplications `a_lo * b_lo`, `a_hi * b_lo`, `a_lo * b_hi`, `a_hi * b_lo` and add the results together, taking care to deal with carry correctly.

Since the problem involves manipulation of the Carry flag, writing this function in C does not produce optimal code (see **10.2.2 An example of APCS register usage: 64-bit integer addition** on page 10-5). You therefore have to code the function in ARM assembly language. The following performs the algorithm:

```
; On entry a1 and a2 contain the 32-bit integers to be multiplied (a, b)
; On exit a1 and a2 contain the result (a1 bits 0-31, a2 bits 32-63)
mul64
    MOV     ip, a1, LSR #16      ; ip = a_hi
    MOV     a4, a2, LSR #16      ; a4 = b_hi
    BIC     a1, a1, ip, LSL #16; a1 = a_lo
    BIC     a2, a2, a4, LSL #16; a2 = b_lo
    MUL     a3, a1, a2           ; a3 = a_lo * b_lo      (m_lo)
    MUL     a2, ip, a2           ; a2 = a_hi * b_lo      (m_mid1)
    MUL     a1, a4, a1           ; a1 = a_lo * b_hi      (m_mid2)
    MUL     a4, ip, a4           ; a4 = a_hi * b_hi      (m_hi)
    ADDS    ip, a2, a1           ; ip = m_mid1 + m_mid2   (m_mid)
    ADDCS   a4, a4, #&10000      ; a4 = m_hi + carry    (m_hi')
    ADDS    a1, a3, ip, LSL #16; a1 = m_lo + (m_mid<<16)
    ADC     a2, a4, ip, LSR #16; a2 = m_hi' + (m_mid>>16) + carry
    MOV     pc, lr
```

**Note** On processors with a fast multiply unit (eg. ARM7TDMI, ARM7DMI) this example can be recoded using the *UMULL* instructions.

The above example code is fine for use with assembly language modules, but to use it from C you need to tell the compiler that this routine returns its 64-bit result in registers. You can do this by making the following declarations in a header file:

```
typedef struct int64_struct
{
    unsigned int lo;
    unsigned int hi;
}
int64;

__value_in_regs extern int64 mul64(unsigned a, unsigned b);
```



# Using the Procedure Call Standards

The above assembly language code and declarations, together with a test program, are in the directory `examples/candasm` as the files `mul64.s`, `mul64.h`, `int64.h` and `multest.c`. To compile, assemble and link these to produce an executable image suitable for `armsd`, copy them to your current directory, and then execute the following commands:

```
armasm mul64.s -o mul64.o -li
armcc -c multest.c -li -apcs 3/32bit
armlink mul64.o multest.o libpath/armlib.32l -o multest
```

where `libpath` is the directory in which the semihosted C libraries reside (eg. the `lib` directory of the ARM Software Toolkit).

**Note** `-li` and `-apcs 3/32bit` can be omitted if `armcc` and `armasm` (and `armsd`, below) have been configured with these defaults.

You can then run `multest` under `armsd`:

- 1 Run `armsd` using the following command:

```
armsd -li multest
```

The following is displayed:

```
A.R.M. Source-level Debugger, version 4.10 (A.R.M.) [Aug 26 1992]
ARMulator V1.20, 512 Kb RAM, MMU present, Demon 1.01, FPE,
Little endian.
Object program file multest
armsd:
```

- 2 Type `go` at the `armsd` prompt.

The following line appears:

```
Enter two unsigned 32-bit numbers in hex eg.(100 FF43D)
```

- 3 Type `12345678 10000001`

The following is displayed:

```
Least significant word of result is 92345678
Most significant word of result is 1234567
Program terminated normally at PC = 0x00008418
0x00008418: 0xef000011 .... : > swi      Angel
armsd:
```

- 4 Type `quit` at the `armsd` prompt.

To confirm that `__value_in_regs` is being used, try removing it from `mul64.h`, recompile `multest.c`, relink `multest`, and rerun `armsd`. This time the answers returned will be incorrect, since the result is no longer being returned in registers, but in a block of memory (in other words, the code now has a bug).

# Using the Procedure Call Standards

---



# 11

## Exception Handling

11.1	Overview	11-2
11.2	Entering and Leaving an Exception	11-5
11.3	The Return Address and Return Instruction	11-6
11.4	Installing an Exception Handler	11-8
11.5	SWI Handlers	11-12
11.6	Interrupt Handlers	11-19
11.7	Reset Handlers	11-25
11.8	Undefined Instruction Handlers	11-26
11.9	Prefetch Abort Handler	11-27
11.10	Data Abort Handler	11-28
11.11	Chaining Exception Handlers	11-29
11.12	Additional Considerations on Thumb-Aware Processors	11-31
11.13	System Mode	11-35

# Exception Handling

## 11.1 Overview

During the normal flow of execution through a user program, the program counter increases sequentially through the address space, with branches to nearby labels or branch-with-links to subroutines.

Exceptions occur when this normal flow of execution is diverted, to allow the processor to handle events generated by internal or external sources. Examples of such events are:

- externally generated interrupts
- an attempt by the processor to execute an undefined instruction

It is necessary to preserve the previous processor status when handling such exceptions, so that execution of the original user program can resume once the appropriate exception routine has completed.

The ARM recognizes seven different types of exception, as shown below:

Exception	Description
Reset	Occurs when the CPU reset pin is asserted. This exception is only expected to occur for signalling power-up, or for resetting as if the CPU has just powered up. It can therefore be useful for producing soft resets.
Undefined Instruction	Occurs if neither the CPU nor any attached coprocessor recognizes the currently executing instruction.
Software Interrupt (SWI)	This is a user-defined synchronous interrupt instruction, which allows a program running in user mode to request privileged operations that run in supervisor mode.
Prefetch Abort	Occurs when the CPU attempts to execute an instruction which has prefetched from an <i>illegal</i> address, ie. an address that the memory management subsystem has determined is inaccessible to the CPU in its current mode.
Data Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
IRQ	Occurs when the CPU's external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear.
FIQ	Occurs when the CPU's external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

Table 11-1: Exception types



## 11.1.1 The vector table

Exception handling is controlled by a *vector table*. This is a reserved area of 32 bytes at the bottom of the memory map with one word of space allocated to each exception type (plus one word currently reserved for handling address exceptions when the processor is configured for a 26-bit address space). This is not enough space to contain the full code for a handler, so the vector entry for each exception type typically contains a branch or load PC instruction to continue execution with the appropriate handler.

## 11.1.2 Use of modes and registers by exceptions

As a rule, an application runs in *user mode*, but the servicing of exceptions requires privileged (ie. non-user mode) operation. An exception changes the processor mode, and this in turn means that each exception handler has access to a certain subset of the banked registers:

- its own `r13` or *Stack Pointer* (`SP_mode`)
- its own `r14` or *Link Register* (`LR_mode`)
- its own *Saved Program Status Register* (`SPSR_mode`)

and, in the case of an FIQ, five more general-purpose registers (`r8_FIQ` to `r12_FIQ`).

Each exception handler must ensure that other registers are restored to their original contents upon exit. This can be done by saving the contents of any registers the handler needs to use onto its stack and restoring them before returning.

**Note** *If you are using Angel or ARMulator, the required stacks are set up for you automatically; otherwise, you must set them up yourself.*

## 11.1.3 Exception priorities

When several exceptions occur simultaneously, they are serviced in a fixed order of priority. Each exception is handled in turn before execution of the user program continues. However, it is not possible for all exceptions to occur concurrently. For instance, the undefined instruction and SWI exceptions are mutually exclusive because they both correspond to particular decodings of the current instruction.

**Table 11-2: The exception vectors** on page 11-4 on shows the exceptions, their corresponding processor modes and their handling priorities.

Placing the Data Abort exception above the FIQ exception in the priority list ensures that the Data Abort is actually registered before the FIQ is handled. The Data Abort handler is entered, but control is then passed immediately to the FIQ handler. Once the FIQ has been handled, control returns to the Data Abort Handler. This means that the data transfer error does not escape detection as it would if the FIQ were handled first.

# Exception Handling

Vector Address	Exception Type	Exception Mode	Priority (1=High, 6=Low)
0x0	Reset	svc	1
0x4	Undefined Instruction	undef	6
0x8	Software Interrupt (SWI)	svc	6
0xC	Prefetch Abort	abort	5
0x10	Data Abort	abort	2
0x14	Reserved	Not applicable	Not applicable
0x18	Interrupt (IRQ)	irq	4
0x1C	Fast Interrupt (FIQ)	fiq	3

Table 11-2: The exception vectors



## 11.2 Entering and Leaving an Exception

### 11.2.1 The processor's response to an exception

When an exception is generated, the processor:

- 1 copies the *Current Program Status Register (CPSR)* into the *Saved Program Status Register (SPSR)* for the mode in which the exception is to be handled. This saves the current mode, interrupt mask and condition flags.
- 2 sets the appropriate CPSR mode bits:
  - a) to change to the appropriate mode, also mapping in the appropriate banked registers for that mode.
  - b) to disable interrupts.  
IRQs are disabled when any other type of exception occurs, and FIQs are also disabled when an FIQ occurs.
- 3 stores the *return address* (PC – 4) in *LR\_mode*.
- 4 sets the PC to the appropriate vector address.  
This forces the branch to the appropriate exception handler.

**Note** *If the application is being run on a Thumb-aware processor, the processor's response is slightly different. See **11.12 Additional Considerations on Thumb-Aware Processors** on page 11-31 for more details.*

### 11.2.2 Returning from an exception handler

To return execution to the place where the exception occurred, the handler must:

- restore the CPSR from *SPSR\_mode*
- restore the PC using the return address stored in *LR\_mode*

It carries out these two operations by performing a data processing instruction with the S flag set, and the PC as destination register. Each exception type requires the use of a different instruction—see **11.3 The Return Address and Return Instruction** on page 11-6 for details.

This method also applies to the Load Multiple instruction (using the ^ qualifier). So if a handler stores:

- all the working registers used when the handler is invoked, and;
- the link register, modified to produce the same effect as the data processing instructions given in **11.3 The Return Address and Return Instruction**

onto, say, a full descending stack, the exception handler may restore the registers and return in one instruction:

```
LDMFD sp!, {r0-r12, pc}^
```

# Exception Handling

---

## 11.3 The Return Address and Return Instruction

The actual value in the PC that causes a return from a handler depends on the exception type. Because of the way in which the ARM loads its instructions, when an exception is taken:

- the PC may or may not be updated to the next instruction to be fetched
- the return address may not necessarily be the next instruction pointed to by the PC

When loading the instructions needed for the execution of a program, the ARM uses a pipeline with a fetch, a decode and an execute stage. There is one instruction in each stage of the pipeline at any time. The PC points to the instruction currently being *fetched*. Since each instruction is one word long, the instruction being decoded is at address (PC – 4) and the instruction being executed is at (PC – 8).

**Note** See **11.12.1 The return address** on page 11-32 for details of the return address on Thumb-aware processors when an exception occurs in Thumb state.

### 11.3.1 Returning from SWI and undefined instruction

The SWI and Undefined Instruction exceptions are generated by the instruction itself, so the PC is not updated when the exception is taken. Therefore, storing (PC – 4) in *LR\_mode* makes *LR\_mode* point to the next instruction to be executed. Restoring the PC from the LR with

```
MOVS pc, lr
```

returns control from the handler.

### 11.3.2 Returning from FIQ and IRQ

After executing each instruction, the CPU checks to see whether the interrupt pins are LOW and whether the interrupt disable bits in the CPSR are clear. As a result IRQ or FIQ exceptions are generated only after the PC has been updated, and consequently storing (PC – 4) in *LR\_mode* causes *LR\_mode* to point two instructions beyond where the exception occurred. When the handler has finished, execution must continue from the instruction prior to the one pointed to by *LR\_mode*. The address to continue from is one word (or four bytes) less than that in *LR\_mode*, so the return instruction is:

```
SUBS pc, lr, #4
```

### 11.3.3 Returning from prefetch abort

If the CPU attempts to fetch an instruction from an illegal address, the instruction is flagged as invalid. Instructions already in the pipeline continue to execute until the invalid instruction is reached, at which point a prefetch abort is generated.

The handler gets the MMU to load the appropriate virtual memory locations into physical memory. Having done this, it must return to the offending address and reload the instruction, which should now load and execute correctly.



Because the PC is not updated at the time the prefetch abort is issued, LR\_ABORT points to the instruction following the one that caused the exception. The handler must therefore return to LR\_ABORT – 4 using:

```
SUBS pc, lr, #4
```

## 11.3.4 Returning from data abort

When a load or store instruction tries to access memory, the PC has been updated. A stored value of (PC – 4) in LR\_ABORT points to the second instruction beyond the address where the exception was generated. Once the MMU has loaded the appropriate address into physical memory, the handler should return to the original, aborted instruction so that a second attempt can be made to execute it. The return address is therefore two words (or eight bytes) less than that in LR\_ABORT, making the return instruction:

```
SUBS pc, lr, #8
```

# Exception Handling

---

## 11.4 Installing an Exception Handler

Any new exception handler must be installed in the vector table. When installation is complete, the new handler executes whenever the corresponding exception occurs.

This installation can take one of the following two forms.

### Branch instruction

This is the simplest method of reaching the exception handler. Each entry in the vector table contains a branch to the required handler routine. However, this method does have a limitation: because the branch instruction only has a range of 32MB, with some memory organizations the branch is unable to reach the handler.

### Load PC instruction

With this method, the PC is forced directly to the handler's address by:

- 1 storing the address of the handler in a suitable memory location (within 4KB of the vector address)
- 2 placing an instruction in the vector that loads the PC with the contents of the chosen memory location

### 11.4.1 Installing the handlers at reset

If your application is standalone (ie. does not rely on the debugger or debug monitor to start program execution), it is possible to load the vector table directly from your assembler reset (or startup) code. If your ROM is at location 0x0 in memory, you can simply have a branch statement for each vector at the start of your code. This could also include the FIQ handler if it is running directly from 0x1c. See **11.6 Interrupt Handlers** on page 11-19.

In this case, the following section of code sets up the vectors if they are located in ROM at address zero. Note that you can substitute branch statements for the loads.

```
Vector_Init_Block
    LDR    PC, Reset_Addr
    LDR    PC, Undefined_Addr
    LDR    PC, SWI_Addr
    LDR    PC, Prefetch_Addr
    LDR    PC, Abort_Addr
    NOP                                ;Reserved vector
    LDR    PC, IRQ_Addr
    LDR    PC, FIQ_Addr

Reset_Addr    DCD    Start_Boot
Undefined_Addr DCD    Undefined_Handler
SWI_Addr      DCD    SWI_Handler
```

```
Prefetch_Addr DCD Prefetch_Handler
Abort_Addr    DCD Abort_Handler
              DCD 0                ;Reserved vector
IRQ_Addr      DCD IRQ_Handler
FIQ_Addr      DCD FIQ_Handler
```

If there is RAM at location zero, the vectors (plus the FIQ handler if required) have to be copied down from an area in ROM into the RAM. In this case, you must use Load PC instructions, and copy the storage locations, to make the code relocatable.

The following piece of code copies down the vectors given above to the vector table in RAM:

```
MOV    r8, #0
ADR    r9, Vector_Init_Block
LDMIA  r9!, {r0-r7}           ;Copy the vectors (8 words)
STMIA  r8!, {r0-r7}
LDMIA  r9!, {r0-r7}           ;Copy the DCD'ed addresses
STMIA  r8!, {r0-r7}           ;(8 words again)
```

For further information, refer to **Chapter 13, Writing code for ROM**.

## 11.4.2 Installing the Handlers from C

Sometimes during development work it is necessary to install exception handlers into the vectors directly from the main application. As a result, the required instruction encoding must be written to the appropriate vector address. This can be done for both the branch and the Load PC method of reaching the handler.

### Branch method

The required instruction can be constructed as follows:

- 1 Take the address of the exception handler.
- 2 Subtract the address of the corresponding vector.
- 3 Subtract 0x8 to allow for the pipeline.
- 4 Shift the result to the right by two to give a word offset, rather than a byte offset.
- 5 Test that the top eight bits of this are clear, to ensure that the result is only 24 bits long (as the offset for the branch is limited to this).
- 6 Logically OR this with 0xea000000 (the opcode for the BAL instruction) to produce the value to be placed in the vector.

# Exception Handling

---

A C function that implements this algorithm is provided below. This takes the following arguments:

- the address of the handler
- the address of the vector in which the handler is to be installed

The function installs the handler and returns the original contents of the vector. This result might be used to create a chain of handlers for a particular exception: see **11.11 Chaining Exception Handlers** on page 11-29 for further details.

```
unsigned Install_Handler (unsigned routine, unsigned *vector)
/* Updates contents of 'vector' to contain branch instruction */
/* to reach 'routine' from 'vector'. Function return value is */
/* original contents of 'vector'.*/
/* NB: 'Routine' must be within range of 32Mbytes from 'vector'.*/
{
    unsigned vec, oldvec;
    vec = ((routine - (unsigned)vector - 0x8)>>2);
    if (vec & 0xff000000)
    {
        printf ("Installation of Handler failed");
        exit (1);
    }
    vec = 0xea000000 | vec;
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
```

Code that calls this to install an IRQ handler might be:

```
unsigned *irqvec = (unsigned *)0x18;
Install_Handler ((unsigned)IRQHandler, irqvec);
```

In this case, the returned, original contents of the IRQ vector are discarded.

## Load PC method

The required instruction can be constructed as follows:

- 1 Take the address of the exception handler.
- 2 Subtract the address of the corresponding vector.
- 3 Subtract 0x8 to allow for the pipeline.
- 4 Logically OR this with 0xe59ff000 (the opcode for LDR PC, [PC,#offset]) to produce the value to be placed in the vector.
- 5 Put the address of the handler into the storage location.

The following C routine implements this:

```
unsigned Install_Handler (unsigned *location, unsigned *vector)
/* Updates contents of 'vector' to contain LDR pc, [pc, #offset] */
/* instruction to cause long branch to address in 'location'. */
/* Function return value is original contents of 'vector'. */
{
    unsigned vec, oldvec;
    vec = ((unsigned)location - (unsigned)vector-0x8) | 0xe59ff000
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
```

Code that calls this to install an IRQ handler might be:

```
unsigned *irqvec = (unsigned *)0x18;
unsigned *irqaddr = (unsigned *)0x38; /* For example */
*irqaddr = (unsigned)IRQHandler;
Install_Handler (irqaddr,irqvec);
```

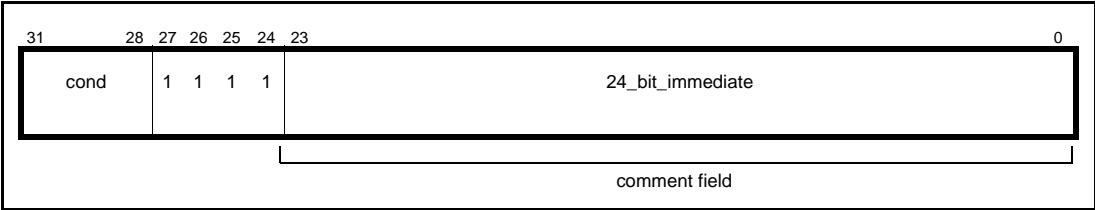
Again in this example the returned, original contents of the IRQ vector are discarded, but they could be used to create a chain of handlers: see **11.11 Chaining Exception Handlers** on page 11-29.

**Note** *If you are operating on a processor with separate instruction and data caches, such as StrongARM-110, you must ensure that cache coherence problems do not prevent the new contents of the vectors from being used. The data cache—or at least the entries containing the modified vectors—must be cleaned to ensure the new vector contents is written to main memory. You must then flush the instruction cache to ensure that the new vector contents is read from main memory. Some operating system environments provide facilities to perform these actions via a SWI IMB (Instruction Memory Barrier). For details of cache clean and flush operations, see the datasheet of the processor you are targeting.*

# Exception Handling

## 11.5 SWI Handlers

When the SWI handler is entered, it must establish which SWI is being called. This information is usually stored in bits 0–23 of the instruction itself, as shown in **Figure 11-1: ARM SWI instruction**.



**Figure 11-1: ARM SWI instruction**

The top-level SWI handler typically accesses the link register and loads the SWI instruction from memory, and therefore has to be written in assembly language. (The individual routines that implement each SWI can be written in C if required.)

The handler must first load the SWI instruction that caused the exception into a register. At this point, LR\_SVC holds the address of the instruction that follows the SWI instruction, so the SWI is loaded into the register (in this case r0) using:

```
LDR r0, [lr,#-4]
```

The handler can then examine the comment field bits, to determine the required operation. The SWI number is extracted by clearing the top eight bits of the opcode:

```
BIC r0, r0, #0xff000000
```

These instructions can be put together to form a top-level SWI handler:

```
AREA TopLevelSwi, CODE, READONLY ; Name this block of code.
EXPORT SWI_Handler
SWI_Handler
    STMFD sp!,{r0-r12,lr}          ; Store registers.
    LDR    r0,[lr,#-4]              ; Calculate address of SWI
                                      ; instruction and load it
                                      ; into r0.
    BIC    r0,r0,#0xff000000        ; Mask off top 8 bits of
                                      ; instruction to give SWI
                                      ; number.

    ;
    ; Use value in r0 to determine which SWI routine to execute.
    ;
    LDMFD  sp!, {r0-r12,pc}^        ; Restore registers and
                                      ; return.
END                                  ; Mark end of this file.
```

**Note** See **11.12.2 Determining the processor state** on page 11-33 for an example of a handler that deals with both ARM- and Thumb-state SWI instructions.

## 11.5.1 SWI handlers in assembly language

The easiest way to call the handler for the requested SWI is to make use of a jump table. If r0 contains the SWI number, the following code could be inserted into the top-level handler given above, following on from the BIC instruction:

```
        ADR r2, SWIJumpTable
        LDR pc, [r2,r0,LSL #2]
SWIJumpTable
        DCD SWInum0
        DCD SWInum1
                                ;
                                ; DCD for each of other SWI routines
                                ;
SWInum0                                ; SWI number 0 code
        B EndofSWI
SWInum1                                ; SWI number 1 code
        B EndofSWI
                                ;
                                ; Rest of SWI handling code
                                ;
EndofSWI
                                ; Return execution to top level
                                ; SWI handler so as to restore registers
                                ; and return to user program.
```

## 11.5.2 SWI handlers in C and assembly language

Although the top-level header must always be written in assembler, the routines that handle each SWI can be written either in assembler or in C.

The top-level header performs a BL (Branch with Link) instruction to jump to the appropriate C function. Since the SWI number is loaded into r0 by the assembler routine, this is passed to the C function as the first parameter (in accordance with the ARM Procedure Call Standard). The function can then use this value in, for example, a switch() statement.

The following line can therefore be added to routine SWI\_Handler:

```
        BL      C_SWI_Handler ; Call C routine to handle the SWI
```

and the C function can be implemented as:

```
void C_SWI_handler (unsigned number)
```

# Exception Handling

---

```
{ switch (number)
{ case 0 : /* SWI number 0 code */
  break;
  case 1 : /* SWI number 1 code */
  break;
  :
  :
  default : /* Unknown SWI - report error */
}
}
```

The code that implements each SWI must be kept as short as possible and, in particular, should not call routines from the C library, because these can make many nested procedure calls which can exhaust the stack space and cause the application to crash. (Supervisor mode typically only has a small amount of stack space available to it.)

It is also possible to pass values in and out of such a handler written in C, provided that the top-level handler passes the stack pointer value into the C function as the second parameter (in r1):

```
MOV    r1, sp          ; Second parameter to C routine...
                        ; ...is pointer to register values.
BL     C_SWI_Handler ; Call C routine to handle the SWI
```

and the C function is updated to access it:

```
void C_SWI_handler (unsigned number, unsigned *reg)
```

The C function can now access the values contained in the registers at the time the SWI instruction was encountered in the main application code (see **Figure 11-2: Accessing the supervisor stack**, below). It can read from them:

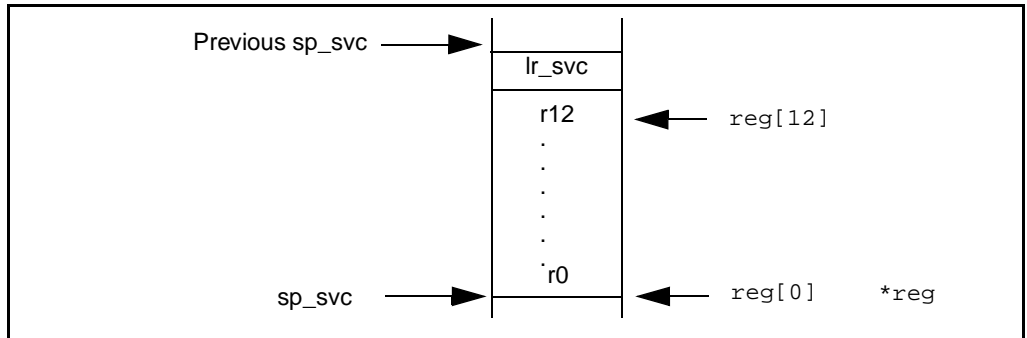
```
value_in_reg_0 = reg [0];
value_in_reg_1 = reg [1];
:
:
value_in_reg_12 = reg [12];
```

and also write back to them:

```
reg [0] = updated_value_0;
reg [1] = updated_value_1;
:
:
reg [12] = updated_value_12;
```

causing the updated value to be written into the appropriate stack position, and then restored into the register by the top-level handler.





**Figure 11-2: Accessing the supervisor stack**

## 11.5.3 Using SWIs in supervisor mode

When a SWI instruction is executed, supervisor mode is entered as part of the exception handling process, the CPSR is copied into `spsr_svc`, and the return address is stored in `lr_svc`, as described in **11.2.1 The processor's response to an exception** on page 11-5. As a result, if the processor is already in supervisor mode, the LR and SPSR will be corrupted, so ensure that the original values of the LR and SPSR are not lost.

This can happen if the handling of a particular SWI number causes a further SWI to be called. Where this occurs, it is possible to modify the SWI handler routine previously given to store SPSR onto the stack in addition to LR. This ensures that each invocation of the handler saves the necessary information to return to the instruction following the SWI that invoked it. The following code will do this:

```
SWI_Handler
    STMFD    sp!, {r0-r12,lr}    ; Store registers.
    LDR      r0,[lr,#-4]         ; Calculate address of SWI
                                ; instruction...
                                ; ...and load it into r0.
    BIC      r0,r0,#0xff000000   ; Mask off top 8 bits of
                                ; instruction to give SWI number.
    MOV      r1, sp              ; Second parameter to C routine...
                                ; ...is pointer to register values.
    MRS      r2, spsr            ; Move the spsr into a gp register.
    STMFD    sp!, {r2}          ; Store spsr onto stack. This is
                                ; only really needed in case of
                                ; nested SWIs.
    BL       C_SWI_Handler      ; Call C routine to handle the SWI.
    LDMFD    sp!, {r2}          ; Restore spsr from stack into r2...
    MSR      spsr, r2           ; ... and restore it into spsr.
```

# Exception Handling

---

```
LDMFD sp!, {r0-r12,pc}^ ; Restore registers and return.  
END                      ; Mark end of this file.
```

A further problem can be encountered if the second level of the handler is written in C. By default, the C compiler does not take into account that an inline SWI may overwrite the contents of `lr`. It is therefore necessary to instruct the C compiler to allow for this using the `-fz` compiler option. So if the C function is in module `c_swi_handle.c`, the following command produces the object code file:

```
armcc -c -fz c_swi_handle.c
```

The `-fz` option can also be used with `tcc`.

**Note** *On processors that implement ARM Architecture 4 or 4T, it is also possible to make use of system mode to avoid this problem. See **11.13 System Mode** on page 11-35 for details.*

## 11.5.4 Calling SWIs from an application

The easiest way to call SWIs from your application code is to set up any required register values and call the relevant SWI in assembler language. For example:

```
MOV    r0, #65    ; load r0 with the value 65  
SWI     0x0        ; Call SWI 0x0 with parameter value in r0
```

The SWI can be conditionally executed, as can all ARM instructions.

Calling a SWI from C is more complicated as it is necessary to map a function call onto each SWI using the `__swi` compiler directive. This allows a SWI to be compiled inline, without additional calling overhead, provided that:

- its arguments (if any) are passed in `r0–r3` only
- its results (if any) are returned in `r0–r3` only

The parameters are passed to the SWI as if the SWI were a real function call. However, if there are between two and four return values, the compiler must be instructed that the return values are being returned in a structure, and the directive `__value_in_regs` must be used. This is because a struct-valued function is usually treated as if it were a void function whose first argument is the address where the result structure should be placed. See the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for further details.

### Example

This example shows a SWI handler which provides SWI numbers `0x0` and `0x1`. SWI `0x0` takes four integer parameters and returns a single result, whereas SWI `0x1` takes a single parameter and returns four results. To declare these use:

```
struct four  
{ int a, b, c, d;  
};
```

```
__swi (0x0) int calc_one (int,int,int,int);  
__swi (0x1) __value_in_regs struct four calc_four (int);
```

They might then be called using:

```
void func (void)  
{ struct four result;  
  int single, res1, res2, res3, res4;  
  single = calc_one (val1, val2, val3, val4);  
  result = calc_four (val5);  
  res1 = result.a;  
  res2 = result.b;  
  res3 = result.c;  
  res4 = result.d;  
}
```

**Note** `__swi` can also be used with `tcc`.

## 11.5.5 Calling SWIs dynamically from an application

In some circumstances it may be necessary to call a SWI whose number is not known until runtime. This situation might occur, for example, when there are a number of related operations that can be performed on an object, and each operation has its own SWI. In such a case, the methods described above are not appropriate.

There are several ways of dealing with this. For example:

- constructing the SWI instruction from the SWI number, storing it somewhere, then executing it
- using a “generic” SWI which takes, as an extra argument, a code for the actual operation to be performed on its arguments. This generic SWI would then decode the operation and perform it.

The second mechanism can be implemented in assembler by passing the required operation number in a register, typically `r0` or `r12`. The SWI handler can then be rewritten to act on the value in the appropriate register. Because some value has to be passed to the SWI in the comment field, it would be possible for a combination of these two methods to be used. For instance, an operating system might make use of only a single SWI instruction and employ a register to pass the number of the required operation. This would then leave the rest of the SWI space available for application-specific SWIs. This method can also be used if the overhead of extracting the SWI number from the instruction is too great in a particular application.

A mechanism is included in the compiler to support the use of `r12` to pass the value of the required operation. Under the ARM Procedure Call Standard, `r12` is the IP register that has a dedicated role only during function call. At other times it may be used as a scratch register, because it is in the code fragment below. The arguments to the generic SWI are passed in

# Exception Handling

---

registers `r0–r3` and values optionally returned in `r0–r3` as described earlier. The operation number passed in `r12` could be, but need not be, the number of the SWI to be called by the generic SWI.

Below is a C fragment that uses a generic, or *indirect* SWI:

```
__swi_indirect(0x80)
    unsigned SWI_ManipulateObject(unsigned operationNumber,
                                   unsigned object,unsigned parameter);

unsigned DoSelectedManipulation(unsigned object,
                                unsigned parameter, unsigned operation)
{ return SWI_ManipulateObject(operation, object, parameter);
}
```

This produces the following code:

```
EXPORT DoSelectedManipulation
DoSelectedManipulation
    0x000000:  e1a0c002    .... : MOV        r12,r2
    0x000004:  ef000080    .... : SWI         0x80
    0x000008:  e1a0f00e    .... : MOV        pc,r14
```

It is also possible to pass the SWI number in `r0` from C using the `__swi` mechanism. For instance, if SWI `0x0` is used as the generic SWI and operation `0` is a character read and operation `1` a character write, the following can be set up:

```
__swi (0) char __ReadCharacter (unsigned op);
__swi (0) void __WriteCharacter (unsigned op, char c);
```

These can then be used in a more friendly fashion by defining the following:

```
#define ReadCharacter () __ReadCharacter (0);
#define WriteCharacter (c) __WriteCharacter (1, c);
```

However, using `r0` in this way means that only three registers are available for passing parameters to the SWI. Usually, if more parameters need to be passed to a subroutine in addition to `r0–r3`, this can be done using the stack. However, stacked parameters are not easily accessible to a SWI handler, because they typically exist on the user mode stack rather than the supervisor stack employed by the SWI handler.

Alternatively, one of the registers (typically `r1`) can be used to point to a block of memory storing the other parameters.

## 11.6 Interrupt Handlers

The ARM processor has two levels of external interrupt, FIQ and IRQ, both of which are level-sensitive active LOW signals into the core. For an interrupt to be taken, the relevant input must be LOW and the disable bit in the CPSR must be clear.

FIQs have higher priority than IRQs in two ways:

- FIQs are serviced first when multiple interrupts occur.
- Servicing a FIQ causes IRQs to be disabled, preventing them from being serviced until after the FIQ handler has re-enabled them (usually by restoring the CPSR from the SPSR at the end of the handler).

The FIQ vector is the last entry in the vector table (at address `0x1c`) so that the FIQ handler can be placed directly at the vector location and run sequentially from that address. This removes the need for a branch and its associated delays, and also means that if the system has a cache, the vector table and FIQ handler may all be locked down in one block within it. This is important because FIQs are designed to service interrupts as quickly as possible. The five extra FIQ mode banked registers enable status to be held between calls to the handler, again increasing execution speed.

**Note** *An interrupt handler should contain code to clear the source of the interrupt.*

### 11.6.1 Interrupt handlers in C

You can set up ARM-compiled C functions as simple interrupt handlers by using the special function declaration keyword `__irq`. This keyword:

- preserves all registers (excluding the floating-point registers) used by the function.

Without `__irq`, the APCS allows certain registers to be corrupted by a function call.

- exits the function by setting the PC to `(LR - 4)` and restoring the CPSR to its original value

The example handler below reads a byte from location `0x80000000` and writes it to location `0x80000004`:

```
__irq void IRQHandler (void)
{
    volatile char *base = (char *) 0x80000000;
    *(base+4) = *base;
}
```

This produces the following code:

```
EXPORT IRQHandler
IRQHandler
0x000000: e92d0003  .-.  : STMDB      r13!,{r0,r1}
0x000004: e3a00102  .... : MOV        r0,#0x80000000
0x000008: e5d01000  .... : LDRB       r1,[r0,#0]
```

# Exception Handling

---

```
0x00000c: e5c01004 .... : STRB      r1,[r0,#4]
0x000010: e8bd0003 .... : LDMIA    r13!,{r0,r1}
0x000014: e25ef004 ..^.: SUBS      pc,r14,#4
```

Compare this to the result of not using `__irq`:

```
EXPORT IRQHandler
IRQHandler
0x000000: e3a00102 .... : MOV      r0,#0x80000000
0x000004: e5d01000 .... : LDRB     r1,[r0,#0]
0x000008: e5c01004 .... : STRB     r1,[r0,#4]
0x00000c: e1a0f00e .... : MOV      pc,r14
```

However `__irq` is typically only suitable for producing single-level C interrupt handlers. Functions declared in this manner cannot call subroutines written in C, because these could corrupt registers that have not been preserved by the top-level routine.

**Note** *C interrupt handlers cannot be produced in this way using tcc. The `__irq` directive is faulted by tcc. This is because tcc can only produce Thumb code, whereas the core is always in ARM state when an interrupt handler is branched to from the vector table.*

## 11.6.2 Interrupt handlers in assembly language

Interrupt handlers are often written in assembly language to ensure that they execute quickly. Below are some examples.

### Single-channel DMA transfer

The following code is an interrupt handler that performs interrupt driven I/O to memory transfers (soft DMA). The code is especially useful as a FIQ handler. It uses the banked FIQ registers to maintain state between interrupts, so this code is best situated at location 0x1c.

```
r8           points to the base address of the I/O device that data is read from.
IOData       is the offset from the base address to the 32-bit data register that is read
              and reading this register disables the interrupt.

r9           points to the memory location to where that data is being transferred.
r10          points to the last address to transfer to.
```

The entire sequence for handling a normal transfer is four instructions. Code situated after the conditional return is used to signal that the transfer is complete.

```
LDR    r11, [r8, #IOData]; Load port data from the IO device.
STR    r11, [r9], #4      ; Store it to memory: update the pointer.
CMP    r9, r10            ; Reached the end ?
SUBLES pc, lr, #4         ; No, so return.
                          ; Insert transfer complete code here.
```

Byte transfers can be made by replacing the load instructions with load byte instructions, and transfers from memory to an I/O device are made by swapping the addressing modes between the load instruction and the store instruction.

## Dual-channel DMA transfer

This example is similar to the one in *Single-channel DMA transfer* on page 11-20, except that here two channels are being handled (which may be the input and output side of the same channel). Again, this code is especially useful as a FIQ handler, because it uses the banked FIQ registers to maintain state between interrupts. For this reason it is best situated at location 0x1c.

r8	points to the base address of the I/O device from which data is read.
IOStat	is the offset from the base address to a register indicating which of two ports caused the interrupt.
IOPort1Active	is a bit mask indicating if the first port caused the interrupt (otherwise it is assumed that the second port caused the interrupt).
IOPort, IOPort2	are offsets to the two data registers to be read. Reading a data register disables the interrupt for the corresponding port.
r9	points to the memory location to which data from the first port is being transferred.
r10	points to the memory location to which data from the second port is being transferred.
r11 and r12	point to the last address to transfer to (r11 for the first port, r12 for the second).

The entire sequence to handle a normal transfer comprises nine instructions; code situated after the conditional return is used to signal that the transfer is complete.

```
LDR      r13, [r8, #IOStat]      ; Load status register to
                                ; find which port caused
TST      r13, #IOPort1Active     ; the interrupt.
LDREQ    r13, [r8, #IOPort1]     ; Load port 1 data.
LDRNE    r13, [r8, #IOPort2]     ; Load port 2 data.
STREQ    r13, [r9], #4           ; Store to buffer 1.
STRNE    r13, [r10], #4         ; Store to buffer 2.
CMP      r9, r11                 ; Reached the end?
CMPL     r10, r12                ; On either channel?
SUBLES   pc, lr, #4              ; Return
; Insert transfer complete code here.
```

# Exception Handling

---

Byte transfers can be made by replacing the load instructions with load byte instructions, and transfers from memory to an I/O device are made by swapping the addressing modes between the conditional load instructions and the conditional store instructions.

## Interrupt prioritization

This code dispatches up to 32 interrupt sources to their appropriate handler routines. Since it is designed for use with the normal interrupt vector (IRQ), it should be branched to from location 0x18.

External hardware is used to prioritize the interrupt and present the high-priority active interrupt in an I/O register.

IntBase	holds the base address of the interrupt controller.
IntLevel	holds the offset of the register containing the highest-priority active interrupt.
r13	is assumed to point to a small full-descending stack.

Interrupts are enabled after ten instructions (including the branch to this code).

The specific handler for each interrupt is entered after a further two instructions (with all registers preserved on the stack).

In addition, the last three instructions of each handler are then executed with interrupts turned off again, so that the SPSR can be safely recovered from the stack.

### Note

*Application Note 30: Software Prioritization of Interrupts (ARM DAI 0030) describes multiple source prioritization of interrupts using software, as opposed to using hardware as described here.*

```

                                ; first save the critical state
SUB lr, lr, #4                 ; Adjust the return address
                                ; before we save it.
STMFD sp!, {lr}               ; Stack return address
MRS r14, SPSR                 ; get the SPSR ...
STMFD sp!, {r12, r14}         ; ... and stack that plus a
                                ; working register too.
                                ; Now get the priority level of the
                                ; highest priority active interrupt.
MOV r12, #IntBase             ; Get the interrupt controller's
                                ; base address.
LDR r12, [r12, #IntLevel]     ; Get the interrupt level (0 to 31).

                                ; Now read-modify-write the CPSR to enable interrupts.
MRS r14, CPSR                 ; Read the status register.
BIC r14, r14, #0x80           ; Clear the I bit
```



```

; (use 0x40 for the F bit).
MSR CPSR, r14          ; Write it back to re-enable
                        ; interrupts and
LDR PC, [PC, r12, LSL #2] ; jump to the correct handler.
                        ; PC base address points to this
                        ; instruction + 8
NOP                    ; pad so the PC indexes this table.

                        ; Table of handler start addresses

DCD Priority0Handler
DCD Priority1Handler
DCD Priority2Handler
.....

Priority0Handler
    STMFD sp!, {r0 - r11} ; Save other working registers.
                        ; Insert handler code here.
    .....
    LDMFD sp!, {r0 - r11} ; Restore working registers (not r12).

; Now read-modify-write the CPSR to disable interrupts.
MRS r12, CPSR          ; Read the status register.
ORR r12, r12, #0x80     ; Set the I bit
                        ; (use 0x40 for the F bit).
MSR CPSR, r12          ; Write it back to disable interrupts.

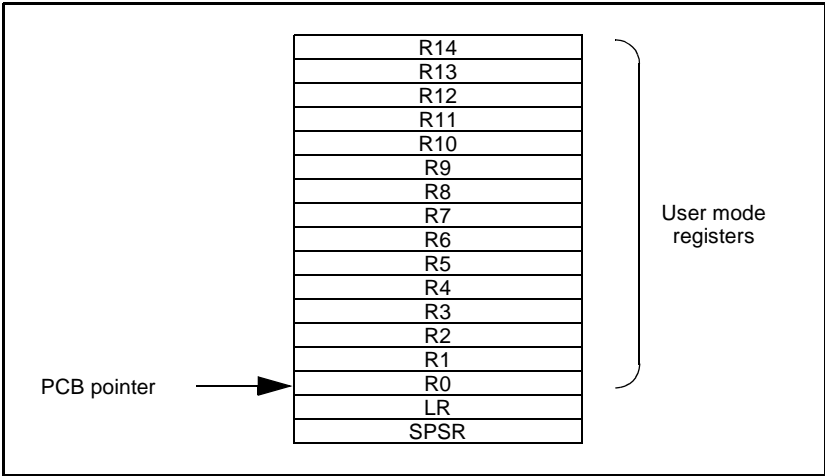
; Now that interrupt disabled, can safely restore SPSR then
return.
LDMFD sp!, {r12, r14}   ; Restore r12 and get SPSR.
MSR SPSR, r14           ; Restore status register from r14.
LDMFD sp!, {PC}^        ; Return from handler.
Priority1Handler
    .....
```

# Exception Handling

## Context switch

This code performs a context switch on the user mode process. The code is based around a list of pointers to *Process Control Blocks (PCBs)* of processes that are ready to run. The pointer to the PCB of the next process to run is pointed to by `r12`, and the end of the list has a zero pointer. Register 13 is a pointer to the PCB, and is preserved between time slices (so that on entry it points to the PCB of the currently running process).

The code assumes the layout of the PCBs shown below in **Figure 11-3: PCB layout**.



**Figure 11-3: PCB layout**

```
STMIA    r13, {r0 - r14}^    ; Dump user registers above r13.
MSR      r0, SPSR             ; Pick up the user status
STMDB    r13, {r0, LR}       ; and dump with return address
                                   ; below.

LDR      r13, [r12], #4       ; Load next process info pointer.
CMP      r13, #0              ; If it is zero, it is invalid
LDMNEDB   r13, {r0, LR}       ; Pick up status and return address.
MRSNE    SPSR, r0             ; Restore the status.
LDMNEIA   r13, {r0 - r14}^    ; Get the rest of the registers
SUBNES    pc, r14             ; and return and restore CPSR.
; Insert "no next process code" here.
```



## 11.7 Reset Handlers

The operations carried out by the Reset handler depend on the system for which the software is being developed. For example, it may:

- set up exception vectors—see **11.4 Installing an Exception Handler** on page 11-8 for details
- initialize stacks and registers
- initialize the memory system (if using an MMU)
- initialize any critical I/O devices
- enable interrupts
- change processor mode and/or state
- initialize variables required by C
- call the main application

For further information, refer to **Chapter 13, Writing code for ROM**.

# Exception Handling

---

## 11.8 Undefined Instruction Handlers

Instructions that are not recognized by the CPU are offered to any coprocessors attached to the system. If the instruction remains unrecognized, an undefined instruction exception is generated. It could be the case that the instruction is intended for a coprocessor, but that the relevant coprocessor—for example a Floating Point Accelerator—is not attached to the system. However, a software emulator for such a coprocessor might be available.

Such an emulator should:

- attach itself to the undefined instruction vector, storing the old contents
- examine the undefined instruction to see if it should be emulated. This is similar to the way in which a SWI handler extracts the number of a SWI, but rather than extracting the bottom 24 bits, the emulator must extract bits 27–24, which determine whether the instruction is a coprocessor operation:
  - If bits 27–24 = b1110 or b110x, the instruction is a coprocessor instruction.
  - If bits 8–11 show that this coprocessor emulator should handle the instruction, the emulator should process the instruction and return to the user program.
  - Otherwise the emulator should pass the exception onto the original handler (or the next emulator in the chain) using the vector stored when the emulator was installed.

Once a chain of emulators is exhausted, no further processing of the instruction can take place, so the undefined instruction handler should report an error and quit. See **11.11 Chaining Exception Handlers** on page 11-29 for more information.

## 11.9 Prefetch Abort Handler

If the system contains no MMU, the Prefetch Abort handler can simply report the error and quit. Otherwise the address that caused the abort must be restored into physical memory. LR\_ABORT points to the instruction at the address following the one that caused the abort, so the address to be restored is at LR\_ABORT – 4. The virtual memory fault for that address can be dealt with and the instruction fetch re-attempted. The handler should therefore return to the same instruction rather than the following one.

# Exception Handling

---

## 11.10 Data Abort Handler

If there is no MMU, the data abort handler should simply report the error and quit. If there is an MMU, the handler should deal with the virtual memory fault.

The instruction that caused the abort is at `LR_ABORT - 8` (since `LR_ABORT` points two instructions beyond the instruction that caused the abort).

Three types of instruction can cause this abort:

- **Single Register Load or Store**
  - If the abort takes place on an ARM6-based core, the following is true:  
If the CPU is in early abort mode and writeback was requested, the address register will not have been updated  
If the CPU is in late abort mode and writeback was requested, the address register will have been updated. The change will need to be undone.
  - If the abort takes place on an ARM7- based core (including ARM7TDMI), the address register will have been updated and the change will need to be undone
  - If the abort takes place on an ARM8 or StrongARM, the address will be restored by the core to the value it had before the instruction started.  
Therefore, no further action is required to undo the change.
- **Swap**  
There is no address register update involved with this instruction.
- **Load / Store Multiple**
  - If the abort takes place on an ARM6- or ARM7-based core, the following is true:  
If writeback is enabled, the base register will have been updated as if the whole transfer had taken place. (In the case of an LDM with the base register in the register list, the processor replaces the overwritten value with the modified base value so that recovery is possible.) The original base address can then be recalculated using the number of registers involved.
  - If the abort takes place on an ARM8 or StrongARM-based core; if writeback is enabled, the base register will be restored to the value it had before the instruction started.

In each of the three cases, the MMU can load the required virtual memory into physical memory: the MMU's *Fault Address Register (FAR)* contains the address that caused the abort. Once this is done, the handler can return and try to execute the instruction again.

## 11.11 Chaining Exception Handlers

In some situations there can be several different sources of a particular exception, for example:

- Angel uses an undefined instruction to implement breakpoints. However, undefined instruction exceptions also occur when a coprocessor instruction is executed, but no coprocessor is present (this mechanism can be used to emulate a coprocessor in software).
- Angel uses a SWI for various purposes, including getting into SVC mode from user mode and supporting semihosting requests. However, an RTOS and/or an application may also wish to implement some SWIs.

In such situations, there are two approaches that can be taken to extend the exception handling code. These are described below.

### 11.11.1 A single extended handler

In some circumstances it is possible to simply extend the code in the exception handler to work out what the source of the exception was, and then directly call the appropriate code. Angel has been written to make this approach simple—Angel already decodes SWIs and undefined instructions, and Angel's exception handlers can be extended to deal with non-Angel SWIs and undefined instructions.

However, this approach is only useful if all the sources of an exception are known when the single exception handler is written.

### 11.11.2 Several chained handlers

Some circumstances require more than a single handler. Consider the situation in which a standard Angel debugger is executing, and a standalone user application (or RTOS) which wants to support some additional SWIs is then downloaded. The newly-loaded application may well have its own entirely independent exception handler that it wants to install, but which cannot simply replace the Angel handler.

In this case, the address of the old handler must be noted, so that the new handler is able to call the old handler if it discovers that the source of the exception is not a source it can deal with. For example, an RTOS SWI handler would call the Angel SWI handler on discovering that the SWI was not an RTOS SWI, so that the Angel SWI handler gets a chance to process it.

This approach can clearly be extended to any number of levels, thus building up a chain of handlers. Note that, although code that takes this approach allows each handler to be entirely independent, it is less efficient than code that uses a single handler, or at least it becomes increasingly less efficient the further down the chain of handlers it has to go.

# Exception Handling

---

Both routines given in **11.4.2 Installing the Handlers from C** return the old contents of the vector automatically. This value can be decoded to give:

- the offset for a branch instruction  
This can be used to calculate the location of the original handler and allow a new branch instruction to be constructed and stored at a suitable place in memory. If the replacement handler fails to handle the exception, it can branch to the constructed branch instruction, which in turn will branch to the original handler.
- the location used to store the address of the original handler  
If the application handler failed to handle the exception, it would then need to load the PC from that location.

In most cases, such calculations may not be necessary, as information on the debug monitor / RTOSs handlers should be available to the application writer. If so, the instructions required to chain in the next handler can be hardcoded into the application. The last section of the application's handler must check that the cause of the exception has been handled. If it has, the handler can return to the application. If not, it will need to call the next handler in the chain.

**Note** *When chaining in a handler before a debug monitor handler, you must remove the chain when the monitor is removed from the system, then directly install the application handler.*



## 11.12 Additional Considerations on Thumb-Aware Processors

**Note** *This section only applies to processors that implement ARM Architecture 4T.*

This section describes the additional considerations you must take into account when writing exception handlers suitable for use on Thumb-aware processors.

Thumb-aware processors use the same basic exception handling mechanism as non Thumb-aware processors, where an exception causes the next instruction to be fetched from the appropriate vector table entry.

The same vector table is used for both Thumb- and ARM-state exceptions. An initial step must be added at the top of the exception handling procedure described in **11.2.1 The processor's response to an exception** on page 11-5. The procedure is:

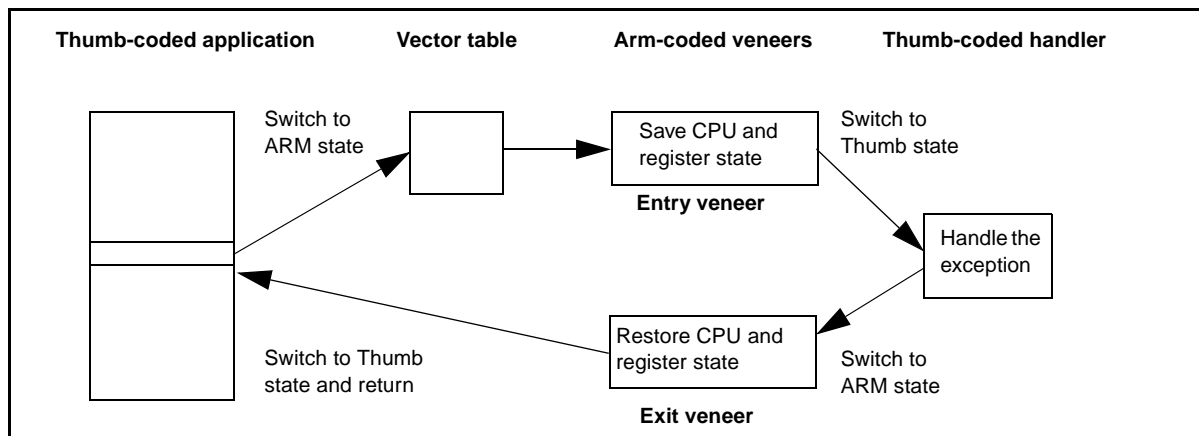
- 1 Check the processor's state. If it is operating in Thumb state, switch to ARM state.
- 2 Copy the CPSR into SPSR\_mode.
- 3 Set the CPSR mode bits.
- 4 Store the return address in LR\_mode.  
See **11.12.1 The return address** on page 11-32 for further details.
- 5 Set the PC to the appropriate vector address.

The switch from Thumb state to ARM state in step 1 ensures that the ARM instruction installed at the appropriate vector (either a branch or a PC-relative load) is correctly fetched, decoded, and executed. Execution then moves to a top-level veneer, also written in ARM code, which saves the processor status and any registers. The programmer then has two choices.

- Write the whole exception handler in ARM code.
- Make the top-level veneer store any necessary status, and then perform a BX (branch and exchange) to a Thumb code routine that handles the exception. Such a routine needs to return to an ARM code veneer in order to return from the exception, since the Thumb instruction set does not have the instructions required to restore the CPSR from the SPSR.

This second strategy is shown in **Figure 11-4: Handling an exception in Thumb state** on page 11-32. See **Chapter 12, Interworking ARM and Thumb** for details of how to combine ARM and Thumb code in this way.

# Exception Handling



**Figure 11-4: Handling an exception in Thumb state**

## 11.12.1 The return address

If an exception occurs in ARM state, the value stored in `LR_mode` is  $(PC - 4)$  as described in **11.3 The Return Address and Return Instruction** on page 11-6. However, if the exception occurs in Thumb state, the processor automatically stores a different value for each of the exception types. This adjustment is required because Thumb instructions take up only a halfword, rather than the full word that ARM instructions occupy.

If this correction were not made by the processor, the handler would have to determine the original state of the processor, and use a different instruction to return to Thumb code rather than ARM code. By making this adjustment, however, the processor allows the handler to have a single return instruction which will return correctly, regardless of the processor's state (ARM or Thumb) at the time the exception occurred.

There follows a summary of the values to which the processor sets `LR_mode` if an exception occurs when the processor is in Thumb state.

### SWI and Undefined Instruction handlers

The handler's return instruction (`MOVS pc, lr`) must reset the PC to the address of the next instruction to execute. This is at  $(PC - 2)$ , so the value stored by the processor in `LR_mode` is  $(PC - 2)$ .

### FIQ and IRQ handlers

The handler's return instruction (`SUBS pc, lr, #4`) must reset the PC to the address of the next instruction to execute. Because the PC is updated before the exception is taken, the next instruction is at  $(PC - 4)$ . The value stored by the processor in `LR_mode` is therefore PC.

## Prefetch abort handlers

The handler's return instruction (`SUBS pc, lr, #4`) must reset the PC to the address of the aborted instruction. Because the PC is not updated before the exception is taken, the aborted instruction is at  $(PC - 4)$ . The value stored by the processor in `LR_mode` is therefore PC.

## Data abort handlers

The handler's return instruction (`SUBS pc, lr, #8`) must reset the PC to the address of the aborted instruction. Because the PC is updated before the exception is taken, the aborted instruction is at  $(PC - 6)$ . The value stored by the processor in `LR_mode` is therefore  $(PC + 2)$ .

### 11.12.2 Determining the processor state

An exception handler may need to determine whether the processor was in ARM or Thumb state when the exception occurred. SWI handlers, especially, may need to read the processor state. This is done by examining the SPSR's T bit, which is set for Thumb state and clear for ARM state.

Both ARM and Thumb instruction sets have the SWI instruction. We have already examined how to handle SWIs called from ARM state (in **11.5 SWI Handlers** on page 11-12). Here we address the handling of SWIs that are called from Thumb state. When doing so there are three considerations to bear in mind:

- the address of the instruction is at (LR-2), rather than (LR-4)
- the instruction itself is 16-bit, and so requires a halfword load
- the SWI number is held in 8 bits instead of the ARM's 24 bits

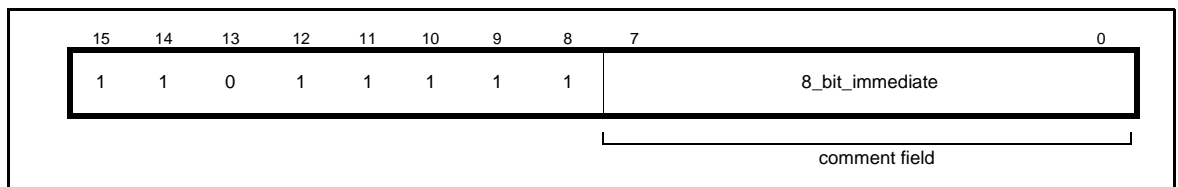


Figure 11-5: Thumb SWI instruction

# Exception Handling

---

## Example

The following ARM code example handles a SWI from both sources:

```
T_bit EQU 0x20; Thumb bit of CPSR/SPSR, ie bit 5.
:
:
SWIHandler
    STMFD sp!, {r0-r12,lr} ; Store the registers.
    MRS r0, spsr           ; Move SPSR into general purpose
                           ; register.
    TST r0, #T_bit         ; Test if bit 5 is set.
    LDRNEH r0,[lr,#-2]      ; T_bit set so load halfword (Thumb)
    BICNE 0,r0,#0xff00      ; and clear top 8 bits of halfword
                           ; (LDRH clears top 16 bits of word).
    LDREQ r0,[lr,#-4]       ; T_bit clear so load word (ARM)
    BICEQ r0,r0,#0xff000000; and clear top 8 bits of word.

    ADR r1, switable        ; Load address of the jump table.
    LDR pc, [r1,r0,LSL#2]; Jump to the appropriate routine.
switable
    DCD do_swi_1
    DCD do_swi_2
    :
    :
do_swi_1
    ; Handle the SWI.
    LDMFD sp!, {r0-r12,pc}^; Restore the registers and return.
do_swi_2
    :
```

## Notes

- 1 Each of the `do_swi_x` routines could carry out a switch to Thumb state and back again to reduce code density if required.
- 2 The jump table could be replaced by a call to a C function containing a `switch()` statement to implement the SWIs.
- 3 It would be possible for a SWI number to be handled differently depending upon the state it was called from.
- 4 The range of SWI numbers accessible from Thumb state can be increased by calling SWIs dynamically as described in **11.5 SWI Handlers** on page 11-12.

## 11.13 System Mode

**Note** *This section only applies to processors that implement ARM Architectures 4 and 4T.*

The ARM Architecture defines a user mode which has 15 general-purpose registers, a PC and a CPSR. In addition to this mode there are five privileged processor modes, each of which have an SPSR and a number of registers that replace some of the 15 user mode general-purpose registers.

When a processor exception occurs, the current program counter is copied into the exception mode's `r14 (lr)`, and the CPSR is copied into the exception mode's SPSR. The CPSR is then altered in an exception-dependent way, and the PC is set to an exception-defined address to start the exception handler.

The ARM subroutine call instruction (`BL`) copies the return address into `r14` before changing the PC, so the subroutine return instruction moves `r14` to the PC (`MOV PC, LR`).

Together these actions imply that ARM modes which handle exceptions must ensure that they do not cause the same type of exceptions if they call subroutines, because if the exception occurs just after a `BL`, the subroutine return address will be overwritten with the exception return address.

In earlier versions of the ARM architecture, this problem has been solved by either carefully avoiding subroutine calls in exception code, or changing from the privileged mode to user mode. The first solution is often too restrictive, and the second means the task may not have the privileged access it needs to run correctly.

ARM Architecture 4 introduces a new processor mode, called *system* mode, to help avoid this problem. System mode is a privileged processor mode that shares the user mode registers. Privileged mode tasks can run in this mode, and exceptions no longer overwrite the link register.

**Note** *System mode cannot be entered via an exception. An exception handler must cause it to be entered by modifying the CPSR.*



# 12

## Interworking ARM and Thumb

12.1	Introduction	12-2
12.2	Basic Assembler Interworking	12-4
12.3	C Interworking and Veneers	12-10
12.4	Assembler Interworking Using Veneers	12-18
12.5	Interworking and the ARM Project Manager	12-21
12.6	Using the Thumb-ARM Interworking Image Project	12-22
12.7	Modifying Project to Support Interworking	12-24
12.8	Library usage and the ARM Project Manager	12-25

# Interworking ARM and Thumb

---

## 12.1 Introduction

This chapter explains *ARM/Thumb interworking*, the procedure used to change from running in ARM state to running in Thumb state, and vice versa. The information in this chapter is only relevant to ARM processors that implement ARM architecture 4T—for example the ARM7TDMI.

ARM and Thumb code can be freely mixed provided that the code conforms to the requirements of the *ARM Procedure Call Standard (APCS)* and the *Thumb Procedure Call Standard (TPCS)* respectively.

Compiled code automatically conforms to these standards. Assembler programmers must ensure that their code also conforms. See **Chapter 10, Using the Procedure Call Standards** for details.

The ARM linker automatically detects when ARM and Thumb code is being mixed, and generates small code segments called *veneers*. These perform an ARM–Thumb state change on function entry and exit whenever an ARM function is called from Thumb state, or a Thumb function is called from ARM state.

When you use a Thumb-aware ARM processor, you will probably write most of your application to run in Thumb state, because this provides the best possible code density and performance when running from 8- or 16-bit memory. However, parts of your application may need to run in ARM state for the reasons explained below.

### Speed

Some parts of an application may be highly speed critical. These sections may be more efficient running in ARM state rather than Thumb state, because in some circumstances a single ARM instruction can do more than the equivalent Thumb instruction. This requirement for speed may mean that there is a small amount of fast 32-bit memory in the system from which ARM code can be run, without the overhead of fetching each instruction from 8- or 16-bit memory.

### Reduced functionality

To improve code density, the 32-bit ARM instruction set was reduced to 16 bits to create the Thumb instruction set. This reduction makes Thumb instructions less flexible than their ARM equivalents. Some operations (for example, accessing the program status registers directly) are not possible in Thumb state. This means that a state change is required before carrying out certain operations.

### Exception handling

When dealing with an exception, the processor automatically enters ARM state. This means that the first part of a handler needs to be coded with ARM instructions, even if it re-enters Thumb state to carry out the main processing of the exception. Note that at the end of such processing, the processor must re-enter ARM state to return from the handler to the main application.

For further details see **Chapter 11, Exception Handling**.



# Interworking ARM and Thumb

---

## Simple standalone Thumb assembler programs

A Thumb-aware processor can power up in ARM state. To run simple Thumb assembler programs under the debugger, add an ARM assembler header which first carries out a state change and then calls the main Thumb routine. See the ***Branch Exchange example*** on page 12-5.

# Interworking ARM and Thumb

---

## 12.2 Basic Assembler Interworking

The simplest method of interworking takes place at the level of hand-coded assembly language. In this case, it is up to the programmer to make sure that the register usage is compatible between any interworking routines.

### 12.2.1 The Branch Exchange instruction

Both the Thumb instruction set and the ARM instruction set (on Thumb-aware cores only) support an instruction known as *Branch Exchange* or **BX**. The syntax of this instruction is:

Thumb	<b>BX</b> <i>Rn</i>
ARM	<b>BXcond</b> <i>Rn</i>

In both cases *Rn* can be any register in the range 0 to 15. Each of these instructions branches to the address contained in the given register. The ARM variant can also be conditionally executed, just like any other ARM instruction.

**Note** *Using a **BX** *pc* (ie. *r15*) in Thumb state is unsafe because the result is unpredictable if the current PC is only halfword-aligned, rather than fully word-aligned.*

#### Implementing the state change

The state change is implemented using the following information:

- All ARM instructions are word-aligned, so bits 0 and 1 of the address of any ARM instruction are always set to zero (because these bits refer to the halfword and byte part of the address).
- All Thumb instructions are halfword-aligned, so bit 0 of the address of any Thumb instruction is always set to zero (because it refers to the byte part of the address).

Therefore, because bit 0 of the address of any ARM or Thumb instruction is always zero, it is possible to use this bit in the **BX** instruction. When the processor executes the **BX** instruction, it examines bit 0 of the register and determines in which state to execute the instruction being branched to:

- If bit 0 is set, the instruction branched to will be executed in Thumb state.
- If bit 0 is clear, the instruction branched to will be executed in ARM state.

This flexibility in specifying the destination state means that you can also use **BX** for branches that do not involve a change of state, thereby allowing you to carry out very long distance branch instructions.

**Note** *This use of a **BX** instruction does not work on non Thumb-aware cores because **BX** does not exist within the ARM instruction set as implemented on such cores. The results of a **BX** on a non Thumb-aware core are unpredictable.*

## Branch Exchange example

The following example program adds together the contents of two registers using Thumb instructions. Before it does this, it must first change the state of the processor from ARM state (the initial state) to Thumb state; it does this using a BX instruction.

Usually, the ARM assembler, `armasm`, is used to assemble ARM code modules, and the Thumb assembler, `tasm`, is used to assemble Thumb code modules. However, you can also mix ARM and Thumb assembly language in a single module if you assemble the code using `tasm`, and prefix the sections of ARM code with the `CODE32` directive and the sections of Thumb code with the `CODE16` directive. For example:

```
AREA AddReg, CODE, READONLY
                                ; Name this block of code.

ENTRY                          ; Mark first instruction to call.
CODE32                         ; Subsequent instructions are ARM.

start
ADR    r2, ThumbProg + 1
                                ; Generate branch target address and set
                                ; bit 0, hence arrive at target in
                                ; Thumb state.
BX     r2                      ; Branch exchange to ThumbProg.

CODE16                         ; Subsequent instructions are Thumb.
ThumbProg
MOV     r2, #2                 ; Load r2 with value 2.
MOV     r3, #3                 ; Load r3 with value 3.
ADD     r2, r2, r3             ; r2 = r2 + r3

stop
MOVr0, #0x18                  ; angel_SWIreason_ReportException
LDRr1, =0x20026                ; ADP_Stopped_ApplicationExit
SWI 0xAB                      ; Angel semihosting Thumb SWI

NOP                                ; Pad out literal pool so
NOP                                ; reachable by LDR
END                                ; Mark end of this file.
```

You can build this module using:

```
tasm -g addreg.s
armlink addreg.o -o addreg
```

# Interworking ARM and Thumb

---

Once built, the module can be loaded into a debugger; set a breakpoint on label `start`, begin execution and when the breakpoint is hit, single step through the rest of the program. If you display the registers after each step, you can see the processor enter and leave Thumb state (as denoted by the “T” in the Current Program Status Register (CPSR) changing from a lowercase “t” to an uppercase “T” and back to the lowercase “t”).

**Note** *By default, `tasm` interprets the instructions as Thumb code when it starts to assemble a module. In the above example, a `CODE32` directive is needed to enable the ARM code header at the start of the module to be correctly interpreted as ARM instructions. You can also specify the default instruction set for `tasm` using the command line options `-16` for Thumb instructions and `-32` for ARM instructions.*

## 12.2.2 Implementing interworking assembler subroutines

To implement a subroutine call in assembler you have to:

- store the return address in the link register
- branch to the address of the required subroutine

In the case of non-interworking subroutine calls, you can carry out both operations in a single `BL` instruction.

In the interworking case (where the subroutine is coded in the other state), you need to allow for state changes both on calling and returning from the subroutine. To call the subroutine and change the state, use a `BX` instruction as already described.

Unlike the `BL` instruction, `BX` does not automatically store the return address in the link register, so you must include a `BX LR` at the end of the subroutine to return to the caller. Remember if you are returning from ARM state to Thumb state, you must also set bit 0 in the link register.

**Note** *You cannot employ the usual `MOV PC, LR` return instruction in this situation because it does not cause the required change of state.*

You can write code to store the link register before branch-exchanging to the subroutine, and, if the call is from Thumb code to ARM code, to set bit 0 in the link register (either in the Thumb caller or the ARM callee). The Thumb instruction set's version of `BL` provides some assistance here, since it sets bit 0 on updating the link register with the return address. When a subroutine returns (using `MOV PC, LR`, or an instruction to load the PC from the stack), bit 0 of the link register is ignored. When a Thumb-to-ARM interworking subroutine call returns using a `BX LR`, it causes the required state change to occur automatically.

So the simplest way to carry out a Thumb-to-ARM interworking subroutine call is to `BL` to an intermediate Thumb code segment, which carries out the `BX` as before. As long as you always use the same register to store the address of the ARM subroutine that is being called from Thumb, this segment can be used to send an interworking call to any ARM subroutine. When carrying out an ARM-to-Thumb interworking subroutine call, you do not need to set bit 0 of the link register, so you can store the return address by copying the program counter into the link register. To do this, add a `MOV LR, PC` instruction immediately before the `BX` instruction.

## Interworking subroutine call examples

### Example 1

The following program has an ARM code header and a Thumb-coded main routine. The program sets up two parameters (r0 and r1), and makes an interworking call to an ARM subroutine that adds the two parameters together and returns.

```
AREA ArmAdd, CODE, READONLY
                                ; name this block of code.
ENTRY                          ; Mark 1st instruction to call.
CODE32                         ; Subsequent instructions are ARM.

start
    ADR    r2, ThumbProg + 1
                                ; Generate branch target
                                ; address and set bit 0, hence
                                ; arrive at target in Thumb
                                ; state.
    BX     r2                  ; Branch exchange to ThumbProg.

                                ; Subsequent instructions are
                                ; Thumb.
ThumbProg
    MOV     r2, #2              ; Load r0 with value 2.
    MOV     r3, #3              ; Load r1 with value 3.
    ADR     r4, ARMSubroutine
                                ; Generate branch target
                                ; address, leaving bit 0 clear,
                                ; in order to arrive in ARM state.
    BL     __call_via_r4        ; Branch and link to Thumb
                                ; code segment that will carry
                                ; out the BX to the ARM
                                ; subroutine. The BL will cause
                                ; bit 0 of LR to be set.

Stop
                                ; Terminate execution.
    MOVR0, #0x18                ; angel_SWIreason_ReportException
    LDRr1, =0x20026              ; ADP_Stopped_ApplicationExit

    SWI 0xAB                    ; Angel semihosting Thumb SWI
__call_via_r4                  ; This Thumb code segment will
                                ; BX to the address contained
                                ; in r4.
```

# Interworking ARM and Thumb

---

```
        BX      r4                ; Branch exchange.

        CODE32                    ; Subsequent instructions are ARM.

ARMSubroutine
        ADD     r0, r0, r1        ; Add the numbers together
        BX      LR                ; and return to Thumb caller
                                   ; (bit 0 of LR set by Thumb BL).

        END                      ; Mark end of this file.
```

You can build the module using:

```
tasm -g armadd.s
armlink armadd.o -o armadd
```

## Example 2

This example is a modified form of the previous example; the main routine is now in ARM code and the subroutine is in Thumb code. Notice that the call sequence is now a MOV instruction followed by a BX instruction.

```
        AREA    ThumbAdd, CODE, READONLY
                                   ; Name this block of code.

        ENTRY   ; Mark 1st instruction to call.
        CODE32  ; Subsequent instructions are ARM.

ArmProg
        MOV     r2, #2            ; Load r0 with value 2.
        MOV     r3, #3            ; Load r1 with value 3.

        ADR     r4, ThumbSub + 1  ; Generate branch target
                                   ; address and set bit 0, hence
                                   ; arrive at target in Thumb state.

        MOV     lr, pc            ; Store the return address.
        BX      r4                ; Branch exchange to subroutine ThumbSub.

Stop
                                   ; Terminate execution.

        MOV     r0, #0x18         ; angel_SWIreason_ReportException
        LDR     r1, =0x20026      ; ADP_Stopped_ApplicationExit
        SWI     0x123456          ; Angel semihosting ARM SWI
```

```
CODE16                ; Subsequent instructions are
                        ; Thumb.

ThumbSub
    ADD    r2, r2, r3    ; Add the numbers together
    BX     LR            ; and return to ARM caller.

END                    ; Mark end of this file.
```

You can build the module using:

```
tasm -g thumbadd.s
armlink thumbadd.o -o thumbadd
```

## 12.2.3 Thumb data in code

When relocating a Thumb code symbol, the linker adds 1 to the value of the symbol so that, if the routine is called using a `BX` instruction, the core is automatically transferred into Thumb state when the routine is called.

However, there is a danger that if you place data within a CODE area in assembler, the linker will add 1 to the value of the symbol because it cannot distinguish between code and data in the CODE area. To allow you to define data within a CODE area, the Thumb assembler provides a `DATA` directive that enables you to mark a symbol as pointing to data within a CODE area.

It is imperative that you use the `DATA` directive when you define data within a Thumb CODE area. This is done as follows:

```
AREAcode, CODE
Thumb_fn    ....
            MOVpc, lr

Thumb_Data  DATA
            DCB1, 3, 4, ...
```

Note that the `DATA` directive must be on the same line as the symbol.

# Interworking ARM and Thumb

---

## 12.3 C Interworking and Veneers

You can freely mix C code compiled for ARM and Thumb, but small code segments called *veneers* are needed between the ARM and Thumb code to carry out state changes. The linker generates veneers automatically when it detects interworking calls.

### 12.3.1 Software stack checking and frame pointers

In the ARM world, both software stack checking and the use of a frame pointer in compiled code are turned on by default. In the Thumb world, frame pointers are not used, and software stack checking is turned off by default (most Thumb-based systems are expected to have hardware stack limit checking). Therefore, you must specify your requirements on every compiler invocation to ensure that all files are compiled to the same standard.

If you don't do this, the linker informs you where the incompatibilities occurred by generating warning messages of the form:

```
Attribute conflict between AREA object(area) and image code.  
(attribute difference = {NO_SW_STACK_CHECK}).
```

To turn off software stack checking (and use of frame pointers) in ARM code, use:

```
armcc -apcs /noswst/nofp  
armasm -apcs 3/noswst
```

To turn on software stack checking in Thumb code, use:

```
tcc -apcs /swst  
tasm -apcs 3/swst
```

### 12.3.2 Compiling code for Interworking

Both the ARM and Thumb C compilers have an option to enable them to compile modules containing routines that can be called by routines running in the other state:

```
tcc -apcs /interwork  
armcc -apcs /interwork
```

Modules compiled for interworking generate slightly (typically 2%) larger code for Thumb and marginally larger code for ARM. For a leaf function, the only change is to replace `MOV pc,lr` with `BX lr`. For non-leaf functions, the Thumb compiler must replace, for example, the single instruction:

```
POP {r4,r5,pc}
```

with the sequence:

```
POP {r4,r5}  
POP {r3}  
BX    r3
```



# Interworking ARM and Thumb

---

This has a correspondingly small effect on performance. For this reason it is not necessary to compile all source modules for interworking, but only those that contain subroutines invoked via interworking calls.

In addition, the option `-apcs/interwork` also sets the interwork attribute for the code area into which the modules are compiled. The linker detects this attribute and inserts the appropriate veneer. These veneers are:

- Eight bytes per called routine for calls from Thumb to ARM.  
These comprise a Thumb `BX` instruction, a halfword of padding, and an ARM branch instruction.
- 12 bytes per called routine for calls from ARM to Thumb.  
These comprise an ARM `LDR` instruction to get the address of the function being called, an ARM `BX` instruction to execute the call, and a word to hold the address of the function.

**Note** *Remember that ARM code compiled for interworking cannot be used on non Thumb-compatible ARMs because the instruction sets used by such cores does not contain the `BX` instruction.*

In this release of the ARM Software Development Toolkit, the code size given by the `armlink -info totals` option does not include the code added automatically by the linker to act as the interworking veneer. (This is also true of other areas such as scatter loading). To find the amount of space taken by the veneers, it is necessary to use the `armlink -MAP` option. The total space occupied by the veneers is the size (in hexadecimal) given for the area `IWV$$Code`.

# Interworking ARM and Thumb

---

## Simple C interworking example

The two modules in the following example can be built to produce an application where `main()` is a Thumb routine that carries out an interworking call to an ARM subroutine. The subroutine call itself makes an interworking call to the Thumb library routine `printf()`.

```
/* *****
 *      thumb.c      *
 * ***** */
#include <stdio.h>
extern void arm_function(void);
int main(void)
{
    printf("Hello from Thumb World\n");
    arm_function();
    printf("And goodbye from Thumb World\n");
    return (0);
}

/* *****
 *      arm.c      *
 * ***** */
#include <stdio.h>
void arm_function(void)
{
    printf("Hello and Goodbye from ARM world\n");
}
```

Compile and link these two modules using:

```
tcc -c -apcs /interwork -o thumb.o thumb.c
armcc -c -apcs /noswst/nofp/interwork -o arm.o arm.c
armlink -o hello thumb.o arm.o armlib_i.161
```

**Note** *If `armlib_i.161` is not in the current directory, you must add the full pathname to it. For more details about compiling `thumb.c` for interworking, and an explanation of `_i`, see **12.3.3 Simple rules for interworking** on page 12-13.*

At this stage you can load the program into a debugger to execute it. If you wish to examine the size of the veneers automatically inserted by the linker, type:

```
armlink -o hello thumb.o arm.o armlib_i.161 -map -list mapfile
```

The `-list` option redirects the output from the linker to the file `mapfile`. You can then load this file into an editor and find the `IWV$$Code` area which gives the total size of the veneers.

```
AREA map of hello:
```

```
Base      Size      TypeRO?Name
```

```
      :      ::      :      :
```

```
a7b0     14      DATARO IWV$$Code from object file <anon>
```

```
      :      ::      :      :
```

For this example the veneer code size is 0x14 (ie. 20) bytes long: eight bytes for the call from `main()` to `arm_function()` and 12 bytes for the call from `arm_function()` to `printf()`.

## 12.3.3 Simple rules for interworking

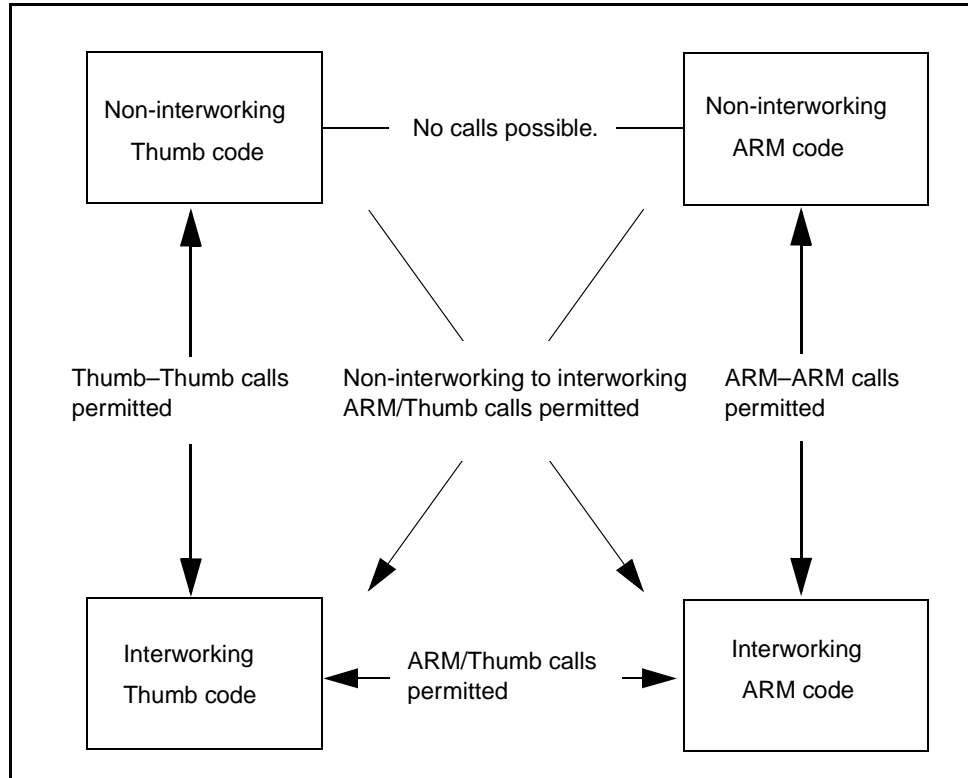
When using interworking within an application, the rules that you need to follow are:

- You must compile any C modules containing functions that are called via interworking calls using the `-apcs /interwork` command-line option.
- You may compile modules that are never called via an interworking call without the `-apcs /interwork` option. These modules may make interworking calls, but may not be called via interworking calls.
- Never make indirect calls (for example, calls using function pointers) to non-interworking code from code in the other state.
- When producing an application that contains interworking calls and linking with the standard ANSI C library, always use an interworking version of the library (ie. `armlib_i.16x`—see **12.3.6 The C library** on page 12-15 for further details). The library used should be the one that matches the state in which the `main()` function executes.

These rules are summarized in **Figure 12-1: Interworking using direct calls** on page 12-14

**Note** *You must take great care when using function pointers in applications that contain both ARM and Thumb code. The linker cannot generate warnings about illegal indirect calls, and the code will fail at runtime.*

# Interworking ARM and Thumb



**Figure 12-1: Interworking using direct calls**

## 12.3.4 Detecting interworking calls

To help determine which routines contain interworking calls, the linker generates a warning when it detects a direct ARM–Thumb interworking call where the called routine is not compiled for interworking. By default, these warnings consist of the number of such calls in each direction. The linker has an option to list details of all such calls within an application.

```
armlink -info interwork
```

This generates output in the form:

```
Interworking call from ARM to Thumb code symbol symbol in object (area)
Interworking call from Thumb to ARM code symbol symbol in object (area)
```

These warnings indicate that an ARM-to-Thumb or Thumb-to-ARM interworking call has been detected from the object module *object* to the routine *symbol*, but the called routine has not been compiled for interworking. You will have to recompile the module containing *symbol* for interworking. Therefore, you should interpret these warning messages as errors.

**Note** *You may find that, under some circumstances, code that produces these warnings appears to run correctly. This is not supported, and will be made illegal in future releases of the toolkit. (The warnings currently given by the linker will become errors.) Do not rely on such code executing correctly, or, even if it appears to, being able to build it using future toolkit releases.*

## 12.3.5 Using two copies of the same function

Occasionally, you may wish to include two functions of the same name, one compiled for ARM and the other for Thumb.

Duplicate definitions can be useful, for example:

- if you have a speed-critical routine in a system with 16-bit and 32-bit memory where the overhead of the interworking veneer would degrade the performance too much.
- if you are using an overlay scheme in which you cannot load the ARM or Thumb version of the required routine while in the current state.  
(It is not possible to enter or exit an overlay segment in Thumb-state.)

The linker does allow duplicate definitions provided that each definition is of a different type (ie. one definition defines a Thumb routine, the other defines an ARM routine), but always gives an error message if there is a duplicate definition of a symbol:

Both ARM & Thumb versions of *symbol* present in image

This is a warning to advise you in case you accidentally include two copies of the same routine. If that is what you intended, you may ignore the warning.

**Note** *When both versions of an identically-named routine are present in an image, and a call is made via a function pointer, it is not possible to determine which version of the routine will be called. Therefore, if you are using function pointers to call such routines, you must compile both routines, and the routine making the call, for interworking.*

## 12.3.6 The C library

Two variants of the Thumb C libraries are provided with the Toolkit, one set compiled for interworking (`armlib_i.16l` and `armlib_i.16b`) and one set not compiled for interworking (`armlib.16l` and `armlib.16b`). Only use the non-interworking set if your application consists solely of Thumb code.

Only a non-interworking variant of the ARM C library is provided (`armlib.32l` and `armlib.32b`). Interworking versions of the ARM library are not supplied. They are typically of little use, since only ARM routines are liable to call ARM library routines. (A Thumb routine running from 16-bit memory is unlikely to want to use an ARM library routine that takes up more memory and takes longer to execute than the Thumb library equivalent.) You can, if you wish, build interworking versions of the ARM library; see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for details of how to rebuild the libraries.

# Interworking ARM and Thumb

---

Remember that if interworking takes place within an application, you must use an interworking main library. See **12.3.3 Simple rules for interworking** on page 12-13.

If you need to select the ARM or Thumb version of a standard C library routine explicitly, or if you want to include both ARM and Thumb versions of a routine, you can force the inclusion of specific modules from a library.

To force inclusion of a library module, put the module's name in parentheses after the library name, making sure that there are no spaces between the library name and the opening parenthesis. You can specify more than one module by separating module names with a comma; make sure there are no spaces in the list of module names.

Examples:

- Forcibly use the ARM version of `strlen()` and take all other routines from the (interworking) Thumb library:  

```
armlink -o prog thumb.o arm.o armlib.32l(strlen.o) armlib_i.16l
```
- Forcibly include both ARM and Thumb versions of all functions starting with `str` and take all other routines from the (interworking) Thumb library:  

```
armlink -o prog thumb.o arm.o armlib.16l(str*) armlib.32l(str*) armlib_i.16l
```

**Note** *On Unix platforms, depending on the command shell you are using, you may need to put the characters (, ) and \* in quotes to enter them on the command line.*

## 12.3.7 Differences between 2.02 and 2.11

Some projects built with the Thumb compiler that built correctly under 2.02 fail under 2.10 with undefined symbols at link time. The undefined symbols reported are:

```
__call_via_r0
__call_via_r1
__call_via_r2
__call_via_r3
__call_via_r4
__call_via_r5
__call_via_r6
__call_via_r7
```

This happens when you have a project that does not use the C library (ie. which has its own runtime code), because under 2.02 the Thumb compiler generated the following code to make a call via a function pointer:

```
MOV    lr, pc
MOV    pc, r<n>
```

where `<n>` is a number from 0 to 7. Under 2.1 the compiler generates the following code:

```
BL     __call_via_r<n>
```

# Interworking ARM and Thumb

---

where the routine `__call_via_r<n>` looks like the following:

```
__call_via_r<n>  
    BX      r<n>
```

This change was made to support interworking of ARM/Thumb applications.

To overcome this problem, incorporate the file `ARM211/Cl/Thumb/call_via.s` into your project.

# Interworking ARM and Thumb

## 12.4 Assembler Interworking Using Veneers

The assembler ARM–Thumb interworking explained in **12.2 Basic Assembler Interworking** on page 12-4 carried out all the necessary intermediate processing, so there was no need for the linker to insert interworking veneers, and no need to set the `INTERWORK` attribute that the linker uses to decide whether it needs to add in the veneer.

### 12.4.1 Assembler-only interworking using veneers

You can write assembler ARM–Thumb interworking to make use of the automatically generated linker veneers. To do this, you write:

- the caller routine just as any non-interworking routine, using a `BL` instruction to make the call
- the callee routine to use a `BX` instruction to return, and set the `INTERWORK` attribute for the area in which it is located

#### Example of assembler interworking using veneers

The following example sets registers `r0` to `r2` to the values 1, 2 and 3 respectively. `r0` and `r2` are set by the ARM code, whereas `r1` is set by the Thumb code. Note the setting of the `INTERWORK` attribute in the `AREA` definition of `thumb.s` and the `BX` instruction rather than the usual `MOV pc, lr` to return.

```
; *****
; arm.s
; *****

AREA Arm, CODE, READONLY      ; Name this block of code.
IMPORT ThumbProg
ENTRY                          ; Mark 1st instruction to call.

ARMProg
MOV    r0, #1                  ; Set r0 to show in ARM code.
BL     ThumbProg               ; Call Thumb subroutine.
MOV    r2, #3                  ; Set r2 to show returned to ARM.
MOV    r0, #0 x 18             ; Terminate execution.
MOV    r0, #0x18               ; angel_SWIreason_ReportException
LDR    r1, =0x20026            ; ADP_Stopped_ApplicationExit
SWI    0xAB                    ; Angel semihosting Thumb SWI
END

; *****
; thumb.s
; *****

AREA Thumb, CODE, READONLY, INTERWORK
```



# Interworking ARM and Thumb

```
                                ; Name this block of code.
CODE16                          ; Subsequent instructions are
EXPORT ThumbProg                ; Thumb.

ThumbProg
    MOV    r1, #2                ; Set r1 to show reached Thumb
                                ; code.
    BX     lr                    ; Return to ARM subroutine.
END                              ; Mark end of this file.
```

You can build these using:

```
armasm -apcs 3/noswst/nofp arm.s
tasm thumb.s
armlink arm.o thumb.o -o count
```

When you load the code into the debugger, you can see that the linker has added the required ARM-to-Thumb interworking veneer. This is contained in locations 0x8094 to 0x809c. (Location 0x809c contains the address of the routine being branch-exchanged to, but with bit 0 set.)

ARMSD: list 0x8080

Arm

```
+0000 0x00008080: 0xe3a00001 ....: mov     r0, #1
+0004 0x00008084: 0xeb000002 ....: bl      0x8094
                                ;(ThumbAdd + 0x4)
+0008 0x00008088: 0xe3a02003 ....: mov     r2, #3
+000c 0x0000808c: 0xef000011 ....: swi     0x11
```

Thumb

```
:      mov     r1, #2
+0002 0x00008092: 0x4770      pG : bx      r14
+0004 0x00008094: 0xe59fc000 ....: ldr     r12, 0x0000809c ;
                                =#ThumbAdd+0x1
+0008 0x00008098: 0xe12fff1c../. : bx     r12
+000c 0x0000809c: 0x00008091.... : muleq   r0, r1, r0
```

**Note**     *The addresses quoted may vary depending on which version of the toolkit you are using.*

# Interworking ARM and Thumb

---

## 12.4.2 C and assembler interworking using veneers

C compiled to run in one state can call assembler designed to run in the other state, and vice versa. To do this, write the caller routine as any non-interworking routine, using a BL instruction to make the call. Then:

- if the callee routine is in C, compile it using  
-apcs /interwork
- if the callee routine is in assembler, set the INTERWORK attribute and return using BX lr.

**Note** *Any assembler code used in this manner must be APCS/TPCS-conformant where appropriate.*

### Example of C and assembler interworking using veneers

```
/*****
*      thumb.c      *
*****/
#include <stdio.h>
extern void arm_function(void);
int main(void)
{
    int i = 1;
    printf("i = %d\n", i);
    arm_function(i);
    printf("And now i = %d\n", i);
    return (0);
}
; *****
; arm.s
; *****

        AREA    Arm, CODE, READONLY, INTERWORK
        ; Name this block of code.

arm_function
        ADD     r0, r0, #4    ; Add 4 to first parameter.
        BX     LR           ; Return
        END
```

You can build these using:

```
tcc -apcs /interwork thumb.c
armasm -apcs 3/noswst arm.s
armlink arm.o thumb.o -o add armlib_i.161
```

## 12.5 Interworking and the ARM Project Manager

The *ARM Project Manager (APM)* (see **Chapter 2, ARM Project Manager**) has a template-based structure. All templates supplied with APM that build executable images can support interworking.

The template 'Thumb-ARM Interworking Image' specifically allows an interworking application to be created. It assumes that `tasm` is used for all assembler sources, and that the assembler directives `CODE16` and `CODE32` are used to switch between Thumb and ARM instruction sets. C sources are compiled using either the `tcc` or `armcc` C compiler as needed.

Additionally, projects created from either the 'ARM Executable Image' or 'Thumb Executable Image' templates may be easily modified to support interworking with Thumb or ARM code respectively. For example, an ARM-only application can easily be made into an ARM-mostly project; a Thumb-only project can easily be made into a Thumb-mostly project.

A Thumb application written only in C which needs to implement exception handlers will, by architectural necessity, have these in ARM assembler code and should probably be created using the 'Thumb executable image' template.

### 12.5.1 Choosing a template

Choose a template as follows:

- 1 Within APM select **New** from the **File** menu.
- 2 In the New dialog select **Project**. The New Project dialog is displayed.
- 3 Select a template from the **Type** box. A descriptions of the template is displayed in the field **Template description** when you make a selection.
- 4 Enter a **Project Name** and a **Directory** in which to create it and click **OK**. An empty project based on the template is created in the directory you specified.
- 5 The project tree view appears. Press \* on the numeric keypad to expand all branches, + to expand the selected branch, - to collapse, or click the mouse on the plus/minus icons in the tree view. Double-clicking on an item toggles expansion.

# Interworking ARM and Thumb

---

## 12.6 Using the Thumb-ARM Interworking Image Project

### 12.6.1 Adding files

- 1 If the file is a C source file, you must first select the appropriate partition before adding the file, otherwise go to step 2.
  - a) If the file should be compiled to Thumb code, select the partition **THUMB-C** then select **Add files to THUMB-C** from the **Project** menu.
  - b) If the file is to be compiled to ARM code, select the partition **ARM-C** and select **Add files to ARM-C** from the **Project** menu.

If you select the root of the project tree view APM will try to determine the appropriate partition for adding files, otherwise it will try to add the files to the selected partition (see **2.7.2 When a file type is associated with multiple partitions** on page 2-29).
- 2 Select the top level of the project and select **Add files to Project** from the **Project** menu.
- 3 In the **Add Files to Project** dialog, find the directory containing the files to be added.
- 4 Select the required file or files and click **Open**.

After adding files you may have to expand branches of the tree to make them visible.

Branches containing subtrees have a + button.

If you added files to the THUMB-C partition, perform the steps described in **12.6.2 Configuring the Thumb compiler for interworking**.

If you added files to the ARM-C partition, perform the steps described in **12.6.3 Configuring the ARM compiler for interworking** on page 12-23.

If you added ARM Assembler files to the ASM-Source partition that do not contain CODE32 directives perform the steps **12.6.4 Configuring the Thumb assembler to read ARM assembler source** on page 12-23.

### 12.6.2 Configuring the Thumb compiler for interworking

Configure the Thumb compiler for interworking as follows:

- 1 In the Project View select the partition **THUMB-C**.
- 2 Select **Tool Configuration for THUMB-C** from the **Project** menu.
- 3 Click **tcc** and **Set**.
- 4 In the Compiler Configuration dialog, select the **Target** tab.
- 5 Ensure the check box for **Arm/Thumb Interworking** in the **APCS3** group is checked.
- 6 Review other APCS3 options. For example, it may be necessary to turn off software stack checking.
- 7 Click **OK** to save the configuration.

## 12.6.3 Configuring the ARM compiler for interworking

- 1 In the Project View, select the partition **ARM-C**.
- 2 Choose **Tool Configuration for ARM-C** from the **Project** menu.
- 3 Click **armcc** and **Set**.
- 4 In the Compiler Configuration dialog, select the Target tab.
- 5 Ensure the check box for **Arm/Thumb interworking** in the APCS3 Qualifiers group is check-marked.
- 6 Review other APCS3 options. For example, it may be necessary to turn off software stack checking.
- 7 Click **OK** to save the configuration.

## 12.6.4 Configuring the Thumb assembler to read ARM assembler source

Since armasm supports neither the CODE16 nor CODE32 assembler directives ARM assembler source can not usually be assembled equally by tasm and armasm.

To avoid changing the Assembler source the Thumb assembler may be configured as though a CODE32 directive is processed before any files are read.

- 1 In the Project View, expand the ASM-Sources partition and select the ARM assembler source, for example, `armer_kerl.s`
- 2 Choose **Tool configuration for armer\_kerl.s** from the **Project** menu.
- 3 Click **asm** and **Set**.
- 4 In the Compiler Configuration dialog select the **Target** tab.
- 5 In the **Initial state** group select the **ARM** radio button.
- 6 Select the **Call Standard** tab.
- 7 Ensure the **APCS3** radio button is selected in the APCS3 Qualifiers group.
- 8 Click **OK** to save configuration.

# Interworking ARM and Thumb

---

## 12.7 Modifying Project to Support Interworking

### 12.7.1 Converting an ARM executable image to an ARM-Thumb interworking project

For each file in the SOURCES partition that should be compiled with tcc rather than armcc:

- 1 Select a C file to be compiled into Thumb code from the **Sources** partition, for example, `foo.c`.
- 2 Choose **Edit** variable for `foo.c` from the **Project** menu.
- 3 In the Edit Variables dialog type `cc` in the **Name** field and `tcc` in the **Value** field.
- 4 Configure the Thumb compiler for interworking for this file:
  - a) Select the C file from step 1.
  - b) Select **Tool Configuration for foo.c** from the **Project** menu.
  - c) Click **cc** and **Set**.
  - d) In the Compiler Configuration dialog, select the **Target** tab.
  - e) Ensure the check box for **Arm/Thumb Interworking** in the APCS3 Qualifiers group is check-marked.
  - f) Modify the other APCS3 options if necessary.
  - g) Press **OK** to save configuration.

**Note** *To revert back to armcc set the **Value** from step 3 to empty string, and perform steps 4 to 6 above but click **Unset**. Be aware that this may remove any other per file options you had set.*

### 12.7.2 Converting a Thumb executable image to a Thumb-ARM interworking project

For each file in the SOURCES partition that should be compiled with armcc rather than tcc:

- 1 Select a C file in the partition **SOURCES** that is to be compiled into ARM code, for example, `foo.c`.
- 2 Select **Edit Variable for foo.c** from the **Project** menu.
- 3 In the Edit Variables dialog enter `cc` in the **Name** field, and `armcc` in the **Value** field.
- 4 Configure the ARM compiler for interworking for this file.
  - a) Select the C file from step 1.
  - b) Choose **Tool Configuration for foo.c** from the **Project** menu.
  - c) Click **cc** and **Set**.
  - d) In the Compiler Configuration dialog, choose the **Target** tab.
  - e) Ensure the check box for **Arm/Thumb Interworking** in the APCS3 Qualifiers group is check-marked.
  - f) Modify the other APCS3 options if necessary.
  - g) Click **OK** to save the configuration.

**Note** *To revert back to tcc, set the **Value** from step 3 to empty string, and perform steps 4 to 6 above but click **Unset**. Be aware that this may remove any other per file options you had set.*

## 12.8 Library usage and the ARM Project Manager

In certain circumstances, you may not require the default ANSI C library (for example, when implementing an RTOS with its own stack and heap management).

- 1 Select the project root.
- 2 Select **Tool Configuration for *project.apj*** from the **Project** menu.
- 3 Select **Armlink** then **Set**.
- 4 In the **Armlink** configuration property sheet, select the **General** tab.
- 5 Ensure the check box **Search standard libraries** is not checked.
- 6 Choose the **Further Information** page.
- 7 In the field **Extra command line arguments**, add any libraries to be linked with.

As mentioned in **12.3.6 The C library** on page 12-15, you may sometimes need to force the inclusion of a specific module from a particular library. To do this when using ARM Project Manager 2:

- 1 Select the project root.
- 2 Choose **Tool configuration for *project.apj*** from the **Project** menu.
- 3 Select armlink, then **Set**.
- 4 In the Linker Configuration dialog, select the **Listings** tab.
- 5 In the field **Extra command line arguments**, add the library modules to be forcibly included, for example, to force the inclusion of `strcpy` and `strcmp`:

```
c:\arm210\lib\armlib.32l(strcpy.o)
```

```
c:\arm210\lib\armlib.32l(strcmp.o)
```

The above can also be written using quotes to override the normal meaning of space as an argument separator:

```
c:\arm210\lib\armlib.32l(strcpy.o strcmp.o)
```

Or a pattern may be used for the name of the modules:

```
c:\arm210\lib\armlib.32l(strc*)
```





# 13

## Writing code for ROM

13.1	Introduction	13-2
13.2	Application Startup	13-3
13.3	The Embedded C Library	13-16
13.4	Troubleshooting Hints and Tips	13-24

# Writing code for ROM

---

## 13.1 Introduction

The first part of this chapter describes how to build a simple ROM image that contains C code without using the Angel Debug Monitor. In many situations, the preferred development route will be to use Angel, but an alternative standalone method of constructing an application is presented here.

This chapter also explains how to make use of the Embedded C library, which not only works with standalone ROM images but which can also be used in conjunction with Angel. If the Embedded C library is used with Angel, some of the functions already provided by Angel must be removed to prevent symbol clashes at link time. The Angel functions that may cause duplication—and which would therefore require removal—are in `angel/suppasm.s`.

## 13.2 Application Startup

One of the main considerations with C code in ROM is the way in which the application initializes itself and starts executing. If there is an operating system present, this does not cause a problem since the application is entered automatically via the `main()` function.

In an embedded system there are a number of ways an image may be entered:

- via the RESET vector at location 0
- at the base address of the image

### Applications entered via the RESET vector

The simplest case is one in which the application ROM is located at address 0 in the address map. The first instruction of your application is then a branch instruction to the real entry point.

### Applications entered at the base address

An application may be entered at its base address in one of two ways:

- The hardware fakes a branch at address 0 to the base address of the ROM.
- On RESET the ROM is mapped to address 0 by the memory management. When the application initializes the MMU, it remaps the ROM to its correct address and performs a jump to the copy of itself running at the correct address.

### 13.2.1 Initialization on RESET

No initialization takes place on RESET, so the entry point must perform some initialization before it can call any C code.

Typically, the initialization code may carry out some or all of the following tasks:

- define the entry point  
The assembler directive `ENTRY` marks the entry point.
- set up exception vectors  
If the ROM is located at address 0, the vectors will consist of a sequence of hard-coded branch instructions to the handler for each exception.  
If the ROM is located elsewhere, the vectors must be dynamically initialized by the initialization code. Some standard code for doing this is shown in the code example in **13.2.2 Example 1: Building a ROM to be entered at its base address** on page 13-5. Refer also to **Chapter 11, Exception Handling** for more information.
- initialize the stack pointer registers

Some or all of the following stack pointers may require initialization depending on which interrupts and exceptions are used:

<code>SP_irq</code>	if interrupt requests are used
<code>SP_fiq</code>	if fast interrupt requests are used

The above must be initialized before interrupts are enabled.

# Writing code for ROM

SP_abt	for data abort handling
SP_und	for undefined instruction handling

Generally, the above two are not used in a simple embedded system. However, you may wish to initialize them for debugging purposes.

SP_svc	must always be initialized
--------	----------------------------

- initialize the memory system  
If your system has an MMU, make sure memory mapping is initialized at this point before interrupts are enabled and before any code is called which might rely on RAM being present at a particular address, either explicitly, or implicitly via the use of stack space.
- initialize any critical I/O devices  
Critical I/O devices are any devices that must be initialized before interrupts are enabled. Typically, these devices must be initialized at this point. If they are not, they may cause spurious interrupts when interrupts are enabled.
- initialize any RAM variables required by the interrupt system  
For example, if your interrupt system has buffer pointers to read data into memory buffers, the pointers must be initialized at this time before interrupts are enabled.
- enable interrupts and change processor mode/state if necessary  
At this stage the processor is in supervisor mode. If your application runs in User mode, change to user mode at this point and initialize the user mode SP register.
- initialize memory required by C code  
The initial values for any initialized variables must be copied from ROM to RAM. All other variables must be initialized to zero.  
If the application uses scatter loading, see **Chapter 14, Placing Code and Data in Memory** for details of how to initialize these areas.  
If the application does not use scatter loading, the code to copy initialized variables to RAM should be of the form:

```
IMPORT |Image$$RO$$Limit|      ; End of ROM code (=start of ROM
                                ; data)

IMPORT |Image$$RW$$Base|       ; Base of RAM to initialize
IMPORT |Image$$ZI$$Base|       ; Base and limit of area
IMPORT |Image$$ZI$$Limit|      ; to zero initialize

LDR    r0, =|Image$$RO$$Limit| ; Get pointer to ROM data
LDR    r1, =|Image$$RW$$Base|  ; and RAM copy
LDR    r3, =|Image$$ZI$$Base|  ; Zero init base => top of
                                ; initialized data

CMP     r0, r1                  ; Check that they are different
BEQ     %1

0 CMP   r1, r3                  ; Copy init data
LDRCC  r2, [r0], #4
STRCC  r2, [r1], #4
```

```
BCC    %0
1  LDR    r1, =|Image$$ZI$$Limit| ; Top of zero init segment
    MOV    r2, #0
2  CMP    r3, r1                    ; Zero init
    STRCC  r2, [r3], #4
    BCC    %2
```

- enter C code

If your application runs in Thumb state, you should change to Thumb state using:

```
ORR     lr, pc, #1
BX      lr
```

It is now safe to call C code provided that it does not rely on any memory being initialized. For example:

```
IMPORT  C_Entry
BL      C_Entry
```

## 13.2.2 Example 1: Building a ROM to be entered at its base address

This example shows how to construct a simple piece of code suitable for running from ROM. In a real example, much more would have to go into the initialization section, but as the initialization process is very hardware-specific, it has been omitted from this example.

The following sections of code may be found in the files `init.s` and `ex.c` in directory `examples/rom`.

The commands necessary to build the image are given at the end of the code.

# Writing code for ROM

---

```
--- init.s -----
;
; The AREA must have the attribute READONLY, otherwise the linker
; will not place it in ROM.
;
; The AREA must have the attribute CODE, otherwise the assembler
; will not let us put any code in this AREA
;
; Note the '|' character is used to surround any symbols which
; contain non standard characters like '!'.
        AREA    Init, CODE, READONLY
; Now some standard definitions...
Mode_USR      EQU    0x10
Mode_IRQ      EQU    0x12
Mode_SVC      EQU    0x1
I_Bit         EQU    0x80
F_Bit         EQU    0x40          ; Locations of various things in our
                                   ; memory system
RAM_Base      EQU    0x10000000    ; 64k RAM at this base
RAM_Limit     EQU    0x10010000
IRQ_Stack     EQU    RAM_Limit    ; 1K IRQ stack at top of memory
SVC_Stack     EQU    RAM_Limit-1024 ; followed by SVC stack
USR_Stack     EQU    SVC_Stack-1024 ; followed by user stack
; --- Define entry point
EXPORT __main
; The symbol '__main' is defined here to ensure
__main                ; the C runtime system is not linked in.
ENTRY
; --- Setup interrupt / exception vectors
IF :DEF: ROM_AT_ADDRESS_ZERO
; If the ROM is at address 0 this is just a sequence
; of branches
        B       Reset_Handler
        B       Undefined_Handler
        B       SWI_Handler
        B       Prefetch_Handler
        B       Abort_Handler
        NOP                                ; Reserved vector
        B       IRQ_Handler
```

```

        B            FIQ_Handler

    ELSE
        ; Otherwise we copy a sequence of LDR PC instructions over the
        ; vectors
        ; (Note: We copy LDR PC instructions because branch instructions
        ; could not simply be copied, the offset in the branch instruction
        ; would have to be modified so that it branched into ROM. Also,
        ; branch instructions might not reach if the ROM is at an address
        ; > 32M).

        MOV          r8, #0
        ADR           r9, Vector_Init_Block
        LDMIA         R9!, {r0-r7}
        STMIA         R8!, {r0-r7}
        LDMIA         R9!, {r0-r7}
        STMIA         R8!, {r0-r7}

        ; Now fall into the LDR PC, Reset_Addr instruction which will
        ; continue execution at 'Reset_Handler'
Vector_Init_Block
        LDR           PC, Reset_Addr
        LDR           PC, Undefined_Addr
        LDR           PC, SWI_Addr
        LDR           PC, Prefetch_Addr
        LDR           PC, Abort_Addr
        NOP
        LDR           PC, IRQ_Addr
        LDR           PC, FIQ_Addr

Reset_Addr          DCD Reset_Handler
Undefined_Addr      DCD Undefined_Handler
SWI_Addr            DCD SWI_Handler
Prefetch_Addr      DCD Prefetch_Handler
Abort_Addr          DCD Abort_Handler
                   DCD 0 ; Reserved vector

IRQ_Addr            DCD IRQ_Handler
FIQ_Addr            DCD FIQ_Handler

    ENDIF

        ; The following handlers do not do anything useful in this
        ; example.
Undefined_Handler
```

# Writing code for ROM

---

```

                B      Undefined_Handler
SWI_Handler
                B      SWI_Handler
Prefetch_Handler
                B      Prefetch_Handler
Abort_Handler
                B      Abort_Handler
IRQ_Handler
                B      IRQ_Handler
FIQ_Handler
                B      FIQ_Handler
                ; The RESET entry point
Reset_Handler
                ; --- Initialize stack pointer registers
                ; Enter IRQ mode and set up the IRQ stack pointer
MOV            r0, #Mode_IRQ:OR:I_Bit:OR:F_Bit ; No interrupts
MSR            CPSR, r0
LDR            r13, =IRQ_Stack; Set up other stack pointers
                                   if necessary
; ...
                ; Set up the SVC stack pointer last and return to SVC mode
MOV            r0, #Mode_SVC:OR:I_Bit:OR:F_Bit ; No interrupts
MSR            CPSR, r0
LDR            r13, =SVC_Stack
                ; --- Initialize memory system
                ; ...
                ; --- Initialize critical IO devices
                ; ...
                ; --- Initialize interrupt system variables here
                ; ...
                ; --- Enable interrupts
                ; Now safe to enable interrupts, so do this and remain in SVC
mode
                MOV     r0, #Mode_SVC:OR:F_Bit  ; Only IRQ enabled
                MSR     CPSR, R0
                ; --- Initialize memory required by C code
IMPORT |Image$$RO$$Limit|  ; End of ROM code (=start of ROM
                           ; data)
IMPORT |Image$$RW$$Base|   ; Base of RAM to initialize
```



```
IMPORT |Image$$ZI$$Base|      ; Base and limit of area
IMPORT |Image$$ZI$$Limit|     ; to zero initialize
LDR    r0, =|Image$$RO$$Limit| ; Get pointer to ROM data
LDR    r1, =|Image$$RW$$Base|  ; and RAM copy
LDR    r3, =|Image$$ZI$$Base|  ; Zero init base => top of
                                ; initialized data

CMP    r0, r1                  ; Check that they are different
BEQ    %1

0  CMP    r1, r3                ; Copy init data
LDRCC  r2, [r0], #4
STRCC  r2, [r1], #4
BCC    %0

1  LDR    r1,=|Image$$ZI$$Limit| ; Top of zero init segment
MOV    r2, #0

2  CMP    r3, r1                ; Zero init
STRCC  r2, [r3], #4
BCC    %2

; --- Now we enter the C code
IMPORT C_Entry
[ :DEF:THUMB
    ORR    lr, pc, #1
    BX     lr
    CODE16                                ; Next instruction will be Thumb
]
BL      C_Entry

; In a real application we wouldn't normally expect to return,
; however in case we do the debug monitor swi is used to halt
; the application.
MOV r0,#0x18
LDR r1,= 0x20026
[ :DEF:THUMB
    SWI 0xAB                                ; Thumb Angel semihosting SWI
    NOP                                    ; If we are in Thumb state
    NOP                                    ; here, need to pad end of code
                                           ; so that literal pool will be
                                           ; greater than 8 bytes away.

    |
    SWI 0x123456                            ; ARM Angel semihosting SWI
]
]
```

# Writing code for ROM

---

END

```
--- ex.c -----
#ifdef __thumb
/*define Angel semihosting SWI to be Thumb one */
#define semiSWI 0xAB
#else
/*define Angel semihosting SWI to be ARM one */
#define semiSWI 0x123456
#endif
/* We use the following Debug Monitor SWIs to write things out
 * in this example
 */
__swi (semiSWI) void _Exit(unsigned op, unsigned except);
#define Exit() _Exit (0x18,0x20026)
__swi (semiSWI) void _writec (unsigned op, constchar *c);
#define writec(c)_writec(0x3,c)
/* various memory regions which may need to be copied or
 * initialized
 */
extern char Image$$RO$$Limit[];
extern char Image$$RW$$Base[];
/* We define some more meaningful names here */
#define rom_data_base Image$$RO$$Limit
#define ram_data_base Image$$RW$$Base
/* This is an example of a pre-initialized variable */
static unsigned factory_id = 0xAA55AA55; /* Factory set ID */
/* This is an example of an uninitialized (or 0 initialized)
variable */
static char display[8][40]; /* Screen buffer */
static const char hex[17] = "0123456789ABCDEF";
static void pr_hex(unsigned n)
{
    int i; for (i = 0; i < 8; i++) {
        WriteC(hex[n >> 28]);
        n <<= 4;
    }
}
```

```
void Write0 (const char *string)
{ int pos = 0
  while (string[pos]!=0)
    WriteC (&string[pos++]);
}
void C_Entry(void)
{
  if (rom_data_base == ram_data_base) {
    Write0("Warning: Image has been linked as an application.\r\n");
    Write0(" To link as a ROM image link with the options \r\n");
    Write0(" -RO <rom-base> -RW <ram-base>\r\n");
  }
  Write0("'factory_id' is at address ");
  pr_hex((unsigned)&factory_id);
  Write0(", contents = ");
  pr_hex((unsigned)factory_id);
  Write0("\r\n");
  Write0("'display' is at address ");
  pr_hex((unsigned)display);
  Write0("\r\n");
  Exit ();
}
```

## To build the ROM image

- 1 Compile the C file `ex.c` with the following command. The compiler will generate one warning which may be ignored.  

```
armcc -c -fc -apcs 3/noswst/nofp ex.c
```

<code>-fc</code>	Tells the compiler to allow the \$ character in variables.
<code>-apcs 3/noswst/nofp</code>	Tells the compiler to omit code which does stack checking (noswst) and not to use a frame pointer (nofp).
- 2 Assemble the initialization code `init.s`.  

```
armasm -apcs 3/noswst init.s
```

<code>-apcs 3/noswst</code>	Tells the assembler that this code is only suitable for use with other code that does not have software stack checking. Code that uses software stack checking generally cannot be mixed with code that does not. The assembler marks the
-----------------------------	---

# Writing code for ROM

object file as containing code that does not perform software stack checking so that the linker can give an error if it is mixed with code that does.

### 3 Build the ROM image using armlink.

```
armlink -o ex1_rom -Bin -RO 0xf0000000 -RW 0x10000000
-First init.o(Init) -Remove -NoZeroPad -Map -Info Sizes
init.o ex.o
```

- Bin Tells the linker to produce a plain binary image with no header. This is the most suitable form of image for putting in ROM.
- RO 0xf0000000 Tells the linker that the ReadOnly or code segment will be placed at 0xf0000000 in the address map. This is the base of the ROM in this example.
- RW 0x10000000 Tells the linker that the ReadWrite or data segment will be placed at 0x10000000 in the address map. This is the base of the RAM in this example.
- First init.o(Init) Tells the linker to place this area first in the image. Note that on Unix systems you may need to put a backslash \ before each bracket.
- Remove Tells the linker to remove any unused code areas. In this example there are no unused areas, however this is a useful option for larger ROM builds.
- NoZeroPad Tells the linker not to pad the end of the image with zeros to make space for the variables. This option should always be used when building ROM images.
- Map Tells the linker to print an AREA map or listing showing where each code or data section will be placed in the address space. The output from the above example is given below.
- Info Sizes These two options tell the linker to output various sorts of information during the link process. Neither of these options are necessary to build the ROM but are included here as an example. The output generated by each option is given below.

Here is the AREA listing for the example code:

AREA map of ex1_rom:				
Base	Size	Type	RO?	Name
f0000000	e4	CODE	RO	!!! from object file init.o
f00000e4	238	CODE	RO	C\$\$code from object file ex.o
f000031c	10	CODE	RO	C\$\$constdata from object file ex.o



```
10000000      4      DATA  RW      C$$data from object file ex.o
10000004     140      ZERO   RW      C$$zidata from object file ex.o
```

This shows that the linker places three code areas at successive locations starting from 0xf0000000 (where our ROM is based) and two data areas starting at address 0x10000000 (where our RAM is based).

-Info Sizes tells the linker to print information on the code and data sizes of each object file along with the totals for each type of code or data.

object file	code size	inline data	inline strings	'const' data	RW data	0-Init data	debug data
init.o	228	0	0	0	0	0	0
ex.o	184	28	356	16	4	320	0
Object totals	412	28	356	16	4	320	0

The required ROM size is the sum of the code size (412), the inline data size (28), the inline strings (356), the const data (16) and the RW data (4). In this example, the required ROM size is 816 bytes. This should be exactly the same as the size of the `ex1_rom` image produced by the linker.

The required RAM size is the sum of the RW data (4) and the 0-Init data (320), in this case 324 bytes. Note that the RW data is included in both the ROM and the RAM counts. This is because the ROM contains the initialization values for the RAM data.

## Running the ROM image

You can now run the ROM image using the ARMulator.

Start up `armsd` by typing `armsd` and enter the following commands at the `armsd`: prompt.

```
getfile ex1_rom 0xf0000000
```

This tells `armsd` to load the `ex1_rom` file at address 0xf0000000 in the ARMulator memory map.

Check that the ROM has indeed been loaded correctly by disassembling the first section of it:

```
l 0xf0000000
```

This produces output like the following, which is a disassembly of the first part of `init.s`. If it produces output that has each word reversed (ie. the word at 0xf0000000 is 0x0080a0e3 instead of 0xe3a08000), there is a problem with endianness. Check that your compiler and `armsd` are both configured for the same endianness.

```
0xf0000000: 0xe3a08000    .... :mov    r8,#0
0xf0000004: 0xe28f900c    .... :add    r9,pc,#0xc
0xf0000008: 0xe8b900ff    .... :ldmia  r9!,{r0-r7}
```

## Writing code for ROM

```
0xf000000c: 0xe8a800ff .... :stmia r8!,{r0-r7}
0xf0000010: 0xe8b900ff .... :ldmia r9!,{r0-r7}
0xf0000014: 0xe8a800ff .... :stmia r8!,{r0-r7}
0xf0000018: 0xe59ff018 .... :ldr pc,0xf0000038 ; = #0xf0000070
0xf000001c: 0xe59ff018 .... :ldr pc,0xf000003c ; = #0xf0000058
0xf0000020: 0xe59ff018 .... :ldr pc,0xf0000040 ; = #0xf000005c
0xf0000024: 0xe59ff018 .... :ldr pc,0xf0000044 ; = #0xf0000060
0xf0000028: 0xe59ff018 .... :ldr pc,0xf0000048 ; = #0xf0000064
0xf000002c: 0xe1a00000 .... :nop
0xf0000030: 0xe59ff018 .... :ldr pc,0xf0000050 ; = #0xf0000068
0xf0000034: 0xe59ff018 .... :ldr pc,0xf0000054 ; = #0xf000006c
0xf0000038: 0xf0000070 ...p :andnv r0,r0,r0,ror r0
0xf000003c: 0xf0000058 ...X :andnv r0,r0,r8,asr r0
```

You can now execute the ROM image. Set the PC to the base of the ROM image, then run it.

```
pc=0xf0000000
```

```
go
```

This produces the following output:

```
'factory_id' is at address 10000000, contents = AA55AA55
```

```
'display' is at address 10000004
```

### 13.2.3 Example 2: Building a ROM to be loaded at address 0

Using the same files as in example 1 (`ex.c` and `init.s`), reassemble the `init.s` file using the following command:

```
armasm -apcs 3/noswst -PD 'ROM_AT_ADDRESS_ZERO SETL {TRUE}' init.s
-PD 'ROM_AT_ADDRESS_ZERO SETL {TRUE}'
```

This option tells the assembler to PreDefine the symbol `ROM_AT_ADDRESS_ZERO` and to give it the logical (or Boolean) value `TRUE`.

The assembler file `init.s` tests this symbol and generates different code depending on whether or not the symbol is set.

If the symbol is set, it generates a sequence of branches to be loaded directly over the vector area at address 0.

If you have not already compiled the C file `ex.c`, do that now (see *To build the ROM image* on page 13-11) then relink the image using the following command:

```
armlink -o ex2_rom -Bin -RO 0 -RW 0x10000000 -First init.o(Init)
-Remove -NoZeroPad -Map -Info Sizes init.o ex.o
```

The only difference between this and the command used in example 1 is that here we use `-RO 0` to specify the ROM is based at address 0.

Load and execute the ROM image under ARMulator/armstd as follows.

```
armstd
getfile ex2_rom 0
pc=0
go
```

## 13.2.4 Example 3: Building a ROM using scatter loading

Examples 1 and 2 are simple examples of scatter loading. Hence they can be modified to use the scatter loading mechanisms provided by the linker. The initialization code `init.s` can be modified to use the scatter loading initialization code (see **14.2 Scatter Loading** on page 14-3)

The `Image$$` symbols used in `ex.c` are bound to the same values as in the non scatter loading case.

The linker command would be modified to be:

```
armlink -o ex3_rom -Bin -Scatter scatdes -First init.o(Init)
                                         -Remove -Map -info Sizes init.o ex.o
```

In this case, the `-o` option creates a subdirectory called `ex3_rom` containing a single binary file called `root`.

Note that `-Scatter` tells the linker not to pad the end of the output binary files with zeros. Hence the `-NoZeroPad` option is not required when using `-Scatter`.

`scatdes` is the scatter loading description file. For example 1, this file would be:

```
ROOT 0xf0000000
ROOT-DATA 0x10000000
```

For example 2, the file would be:

```
ROOT 0x0
ROOT-DATA 0x10000000
```

The procedure for running the scatter loaded versions of these examples is identical to the procedure for running non scatter loaded versions, except that the filename used in the `armstd getfile` commands is `ex3_rom/root`.

In these cases, scatter loading does not offer any advantage over the non scatter loading case. However, if you have more than one execute region that needs to be initialized, scatter loading is the easiest method to use.

# Writing code for ROM

---

## 13.3 The Embedded C Library

It is possible to link the standard ANSI C library into an embedded system. However, you may not wish to do this for the following reasons:

- The standard ANSI C library relies on underlying debug monitor SWIs for its operation. Unless your embedded system supports these SWIs in its SWI handler, the C library will not work correctly.
- For the standard ANSI C library to execute, the memory system must be configured in the way expected by the C library. This may not be easy to support in your Embedded system.
- There is a minimum overhead of about 10KB when the standard ANSI C library is included.
- The standard ANSI C library as distributed in binary format is non-reentrant. This may cause problems in embedded systems.

For these reasons ARM has developed the Embedded C library that is a subset of the full ANSI C library but which addresses the issues described above. Only functions that fulfil the following criteria have been included in the Embedded C Library.

- The functions do not make any use of static data. This means that, when using the Embedded C Library, you do not need to worry about issues such as static data re-entrancy within the C library. Because the Embedded C Library does not use any static data, it is automatically fully re-entrant.
- The functions do not rely on the underlying operating system (OS) in any way. Standard functions such as `printf()` and `fopen()` in the full ANSI C library rely on the underlying OS to perform functions such as writing characters, or opening files. All such functions are excluded from the Embedded C library. Although this means that functions such as `printf()` are excluded from the Embedded C library, functions such as `sprintf()` are not, because they do not rely on any underlying OS.
- The functions are relatively stand alone.

Many functions in the full ANSI C library rely on a number of other functions in the C library to perform their operations. For example, `printf()` relies on functions such as `ferror()` and `fputc()` and various functions from `locale.h` and `ctype.h` and also many parts of the software floating point library. This means that including a call to `printf()` means you must include large amounts of the C library. The Embedded C library has been designed to break many of these dependencies so that only the minimum of code needed to perform the operation is included.

The functions in the Embedded C library can be divided into the following three categories:

- The runtime support functions.  
These functions carry out operations, such as integer division, that cannot be performed directly by compiled C code. because the compiler cannot generate code to perform these operations itself, it instead generates calls to functions that can. These functions are provided by the Embedded C library.



- The software floating-point library.

When compiling code for use with software floating-point, the compiler generates calls to routines in the library to perform the floating-point operations that cannot be performed directly in ARM code. For example, to perform a double multiply the compiler generates a call to a routine `_dmul`. All such routines are provided as standard by the Embedded C library. A complete list of these routines is described in **Chapter 15, Floating-Point Support**.

- The C library subset.

This provides a subset of the C library routines which meet the above criteria. A complete list of this subset is in **13.3.3 C library subset** on page 13-20.

## 13.3.1 Embedded C library variants

The following variants of the Embedded C library are provided as standard in the release:

<code>embedded/armlib.16l</code>	Thumb little-endian interworking version with no software stack check and no frame pointer.
<code>embedded/armlib.16b</code>	Thumb big-endian interworking version with no software stack check and no frame pointer.
<code>embedded/armlib_cn.32l</code>	ARM little-endian non-interworking version with no software stack check and no frame pointer.
<code>embedded/armlib_cn.32b</code>	ARM big-endian non-interworking version with no software stack check and no frame pointer.

If you wish to build different variants of the Embedded C library (for example, to add software stack checking), see *Rebuilding the C Library* in the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041). The directories `common/cl/embed_a` and `common/cl/embed_t` contain the make options for the ARM and Thumb libraries respectively. These directories should be specified in place of `targetdir` in the build instructions.

## 13.3.2 Callouts from the Embedded C library

Because the Embedded C library is designed to be completely independent of static data, or any OS-specific calls, it cannot perform operations that are OS-specific or which reference static data directly. Instead it performs a callout to a user-supplied function to perform these operations. There are three callouts that the Embedded C library may make:

<code>__rt_trap</code>	Called when an exception is generated in the Embedded C Library.
<code>__rt_errno_addr</code>	Called by the Embedded C library to get the address of the variable <code>errno</code> .
<code>__rt_fp_status_addr</code>	Called by the Embedded C library to get the address of the floating-point status word.

In each case, the Embedded C library tests for the existence of the callout function before calling it. If the function does not exist some default action is taken. The callout functions along with their default actions are described individually below.

# Writing code for ROM

---

## `__rt_trap`

The Embedded C library does not provide code to handle exceptions such as division by zero. Instead it performs a callout to a user routine `__rt__trap` to handle the exception. The definition of `__rt_trap` is as follows:

```
typedef struct ErrBlock {
    unsigned ErrCode;
    char ErrString[240];
} ErrBlock;
typedef unsigned RegSet[16];
void __rt_trap(ErrBlock *err, RegSet *regs);
```

`ErrBlock` is a block containing an error code followed by a zero-terminated string describing the error.

`RegSet` is a block of 16 words containing the registers at the time of the exception.

The following error codes may be generated by the Embedded C library:

```
0x80000020    Integer divide by zero
0x80000200    Invalid floating point operation
0x80000201    Floating point overflow
0x80000202    Floating point divide by zero
```

By default, if `__rt_trap` is not defined, the Embedded C library executes an undefined instruction.

## `__rt_errno_addr`

Whenever the Embedded C library wishes to read or write `errno` it calls this function to obtain the address of `errno`.

The definition of `__rt_errno_addr`:

```
volatile int *__rt_errno_addr(void);
```

The Embedded C library may set `errno` to one of the following values:

```
EDOM    1    Input argument domain error
ERANGE  2    Result range error
```

By default, if `__rt_errno_addr` is not defined the Embedded C library does not attempt to set or read `errno`.

## `__rt_fp_status_addr`

This function returns the address of the floating point status register.

The definition of `__rt_fp_status_addr`:

```
/*
Descriptions of these bits may be found on page 9-8 of the 7500FE
data sheet. The software floating point library does not implement
the IXE or UFE bits
*/

#define IOC_Bit (1 << 0)      /* Invalid operation cumulative */
#define DZC_Bit (1 << 1)      /* Divide zero cumulative */
#define OFC_Bit (1 << 2)      /* Overflow cumulative */
#define UFC_Bit (1 << 3)      /* Underflow cumulative */
#define IXC_Bit (1 << 4)      /* Inexact cumulative */
#define ND_Bit (1 << 8)       /* No denormalised numbers */
#define NE_Bit (1 << 9)       /* NaN exception bit */
#define SO_Bit (1 << 10)      /* Synchronous operation */
#define EP_Bit (1 << 11)      /* Expanded packed decimal */
#define AC_Bit (1 << 12)      /* Alternative carry */
#define IOE_Bit (1 << 16)     /* Invalid operation exception */
#define DZE_Bit (1 << 17)     /* Divide zero exception */
#define OFE_Bit (1 << 18)     /* Overflow exception */
#define UFE_Bit (1 << 19)     /* Underflow exception */
#define IXE_Bit (1 << 20)     /* Inexact exception */
#define FP_SW_LIB0x40000000
#define FP_SW_EMU0x01000000
#define FP_HW_FPA0x81000000

/* This enables all the exception bits. It is probably a good idea
to have them enabled by default as to do otherwise is not very
friendly.
*/

static unsigned long __fp_status_flags
FP_SW_LIB+IOE_Bit+DZE_Bit+OFE_Bit+UFE_Bit+IXE_Bit;
unsigned long *__rt_fp_status_addr(void)
{
    return &__fp_status_flags;
}
```

# Writing code for ROM

---

By default, if `__rt_fp_status_addr` is not defined the Embedded C library does not attempt to read or write the floating-point status register. For the purposes of raising exceptions, it assumes the invalid operation (IOE), divide by zero (DZE) and overflow (OVE) exception enable bits are set.

## 13.3.3 C library subset

The following C library functions are supported in the Embedded C library.

<code>math.h</code>	<code>fmod, sqrt, fabs, floor, ceil, modf, frexp, ldexp, exp, log, log10, sin, cos, tan, atan, atan2, asin, acos, sinh, cosh, tanh, pow</code>
<code>stdlib.h</code>	<code>abs, div, labs, ldiv, bsearch, qsort, strtol, strtoul, atoi, atol</code>
<code>ctype.h</code>	<code>isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper</code>
<code>string.h</code>	<code>memcmp, memset, memcpy, memmove, strcmp, strncmp, strcpy, strncpy, strlen, memchr, strchr, strcspn, strpbrk, strrchr, strspn, strstr, strcat, strxfrm</code>
<code>stdio.h</code>	<code>sprintf, vsprintf</code>
<code>setjmp.h</code>	<code>setjmp, longjmp</code>

## 13.3.4 Example

The following shows a simple example of using the Embedded C library to format and print a string using the Armulator.

Before any C code can be called, some simple startup code to initialize the system is needed. For the ARMulator all that is required is to initialize the stack pointer.

```
--startup.s-----  
  
AREA  asm_code, CODE  
; If assembled with TASM the variable {CONFIG} will be set to 16  
; If assembled with ARMASM the variable {CONFIG} will be set to 32  
; Set the variable THUMB to TRUE or false depending on whether the  
; file is being assembled with TASM or ARMASM.  
GBLL  THUMB  
[ {CONFIG} = 16  
THUMB SETL  {TRUE}  
; If assembling with TASM go into 32 bit mode as the Armulator will  
; start up the program in ARM state.  
CODE32
```

```
|
THUMB SETL  {FALSE}
]

IMPORT C_Entry
ENTRY
    ; Set up the stack pointer to point to the 512K.
    MOV     sp, #0x80000
    ; Get the address of the C entry point.
    LDR     lr, =C_Entry
    [ THUMB
    ; If building a Thumb version pass control to C_entry using the BX
    ; instruction so the processor will switch to THUMB state.
    BX      lr
    |
    ; Otherwise just pass control to C_entry in ARM state.
    MOV     pc, lr
    ] END
```

Next is the C code to print 10 strings using `sprintf`.

```
--print.c-----

#include <stdio.h>
#ifdef __thumb
/*define Angel semihosting SWI to be Thumb one */
#define semiSWI 0xAB
#else
/*define Angel semihosting SWI to be ARM one */
#define semiSWI 0x123456
#endif
/* We use the following Debug monitor SWIs in this example
*/
__swi (semiSWI) void _Exit(unsigned op, unsigned except);
#define Exit() _Exit (0x18,0x20026)
__swi(semiSWI) void __writeC(unsigned op, const char *C);
#define WriteC(c) __WriteC(0x3,c)
void Write0(const char *string)
{ int pos=0;
```

# Writing code for ROM

---

```
while(string[pos]!=0)
    WriteC(&string[pos++]);
}
int C_Entry(void)
{
    int i;
    char buf[20];
    for (i = 0; i < 10; i++) {
        sprintf(buf, "Hello, World %d\n", i);
        Write0(buf);
    }
    Exit();
}
```

Now, compile and link the program:

```
tasm startup.s
tcc -c print.c
armlink -info totals -o print startup.o print.o directory/
                                                embedded/armlib.16l
```

Where *directory* is replaced with the location of the library directory in your installation.

The linker displays the following output:

	code size	inline data	inline strings	'const' data	RW data	0-Init data	debug data
Object totals	52	0	20	0	0	0	68
Library totals	2452	12	68	0	0	0	224
Grand totals	2504	12	88	0	0	0	292

The exact sizes output by the linker may vary slightly depending on the version of the compiler used.

Now run the program:

```
armsd print
```

At the armsd prompt, enter `go`

The program runs and displays the following output:

```
Hello, World 0
Hello, World 1
Hello, World 2
Hello, World 3
Hello, World 4
```



```
Hello, World 5  
Hello, World 6  
Hello, World 7  
Hello, World 8  
Hello, World 9
```

# Writing code for ROM

---

## 13.4 Troubleshooting Hints and Tips

### 13.4.1 Problem

The linker reports one of the symbols `__rt_stkovf_split_big` or `__rt_stkovf_split_small` as being undefined.

#### Cause

You have compiled your C code with stack checking enabled. The C compiler generates code that calls one of the above functions when stack overflow is detected.

#### Solution

- Recompile your C code with the `-apcs 3/noswst` option to disable stack checking.
- Link with a C library that provides support for stack limit checking

**Note:** This is usually possible only in an application environment because C library stack overflow handling code relies heavily on the application environment.

- Write a pair of functions `__rt_stkovf_split_big` and `__rt_stkovf_split_small`, the code for which usually generates an error for debugging purposes.

The code might look similar to the following:

```
EXPORT __rt_stkovf_split_big
EXPORT __rt_stkovf_split_small
__rt_stkovf_split_big
__rt_stkovf_split_small
    ADR        R0, stack_overflow_message
    SWI        Debug_Message ; System dependent SWI to
                                ; write a debugging message
    B          .              ; and loop forever.
stack_overflow_message
DCB          "Stack overflow", 0
```

### 13.4.2 Problem

The linker generates an error similar to the following:

```
ARM Linker: (Warning) Attribute conflict between AREA
test2.o(C$$code) and image code.
ARM Linker: (attribute difference = {NO_SW_STACK_CHECK}).
ARM Linker: (Warning) Attribute conflict within AREA C$$code
(conflict first found with test2.o(C$$code)).
ARM Linker: (attribute difference = {NO_SW_STACK_CHECK}).
```



## Cause

Parts of your code have been compiled or assembled with software stack checking enabled and parts without. Alternatively, you have linked with a library that has software stack checking enabled whereas your code has it disabled, or vice versa.

## Solution

Make sure all your code is compile assembled with either `-apcs 3/noswst` or `-apcs 3/swst`.

Link with a library built with the same option.

### 13.4.3 Problem

The linker reports `__main` as being undefined.

## Cause

When the compiler compiles the function `main()`, it generates a reference to the symbol `__main` to force the linker to include the basic C runtime system from the semihosted C library. If you are not linking with a semihosted C library and have a function `main()` you may get this error.

## Solution

This problem may be fixed in one of the following ways:

- If the `main()` function is used only when building an application version of your ROM image for debugging purposes, you should comment it out with a `#ifdef` when building a ROM image.
- To avoid confusion, when building a ROM image call the C entry point something other than `main()`, such as `C_Entry` or `ROM_Entry`.
- If you do need a function called `main()`, define a symbol `__main` in your ROM initialization code.

Usually this is defined to be the entry point of the ROM image, so you should define it just before the `ENTRY` directive as follows:

```
EXPORT __main
__main
ENTRY
```

# Writing code for ROM

---

## 13.4.4 Problem

The linker reports a number of undefined symbols of the form:

`__rt...` or `__16__rt...`

### Cause

These are runtime support functions which are called by compiler-generated code to perform tasks that cannot be performed simply in ARM or Thumb code (eg. integer division or floating point operations).

For example, the following code generates a call to runtime support function `__rt_sdiv` to perform a division.

```
int test(int a, int b)
{
    return a / b;
}
```

### Solution

Link with an Embedded C library so that these functions are defined.

## 13.4.5 Problem

The linker produces the error message:

```
ARM Linker: (Fatal) No entry point for image.
ARM Linker: garbage output file aif removed
```

### Cause

You have not defined an entry point. You must define the entry point, even if the entry point is the start of the ROM image.

### Solution

To define an entry point, use the assembler's `ENTRY` directive as shown in the example file `init.s` (see page 12-7).

## 13.4.6 Problem

The compiler produces errors of the form:

```
Serious error: illegal character (0x24 = '$') in source
```

### Cause

The ANSI standard does not permit the `$` character in variable names, although many compilers allow this.

## Solution

Use the `-fc` option on the C compiler to tell it to allow `$` in variable names.

### 13.4.7 Problem

When loading an image into the ARMulator and trying to run it, the following error occurs:

```
*** Error: Can't go
```

## Cause

armsd does not know the location at which it should begin executing your image.

## Solution

Tell armsd where to start executing using the command:

```
pc = address_in_hex
```

Re-enter the `go` command.

If your image is to be executed from its base address, the address you specify above should be the same address as that used in the `getfile` command with which you loaded the image.

### 13.4.8 Problem

The image is bigger than expected (bigger than the size given by `-Info Sizes`).

This problem may also be caused by the image having a large section of zeros on the end of it.

## Cause

By default, when generating a plain binary image, the linker expands zero-initialized areas with zero bytes in the image.

The area is then zero-initialized when the image is loaded directly into memory.

## Solution

Use the `-NoZeroPad` option to tell the linker not to expand the zero init area.

### 13.4.9 Problem

The image compiles and links without problem, but when loaded and disassembled from the base address, no initialization code is present.

## Causes

If the hex words look as though they are reversed instruction words, armsd may be using the wrong endianness.

# Writing code for ROM

---

## Solution

- Reconfigure your copy of `armsd` to the opposite endianness and try again.  
You may have linked it as an application image instead of a plain binary image.  
This is the case if the disassembly looks something like the following:

```
0x10000000: 0xe1a00000 .... : nop
0x10000004: 0xe1a00000 .... : nop
0x10000008: 0xeb00000c .... : bl      0x10000040
0x1000000c: 0xeb00001b .... : bl      0x10000080
0x10000010: 0xef000011 .... : swi     0x11
```

- Relink with the `-bin` flag and without any `-aif` flag.  
The initialization code may not be at the start of the image because you have omitted the `-First` option.
- Try relinking with the `-First` option to see if this resolves the problem.

## 13.4.10 Problem

The image loads without problem but when trying to run, it crashes/hangs immediately.

### Causes

The cause of the previous problem may also apply here.

Another possibility is that the image has been linked or loaded at the wrong address.

### Solution

Check that the address is the same on each of the following:

- The linker's `-RO` option
- The `GetFile` command in `armsd`
- The `PC=` command in `armsd`

If all this is correct, try setting the PC to the start and using the `step in` command to step through all the initialization code to see if it is going wrong in the initialization introduction.

# 14

## Placing Code and Data in Memory

14.1	Introduction	14-2
14.2	Scatter Loading	14-3
14.3	Overlays	14-15
14.4	Overlays using Scatter Loading	14-28

# Placing Code and Data in Memory

---

## 14.1 Introduction

Two of the formats that can be produced by the ARM linker are:

Scatter loading format	Scatter loading format allows you to partition your program image into regions that can be positioned independently in memory. The linker generates the symbols necessary to allow the regions to be loaded from memory at addresses different to their execution addresses (see <b>14.2 Scatter Loading</b> on page 14-3). This format can be selected from the command line using the <code>-SCF</code> and <code>-SCATTER</code> options
ARM Overlay Format	ARM overlay format is a root segment written in <i>ARM Image Format (AIF)</i> together with a collection of overlay segments, each written as a plain binary file. (See <b>14.3 Overlays</b> on page 14-15 for details). A system of overlays may be static (with each segment bound to a fixed address at link time), or dynamic (with each segment relocatable on loading). This format can be selected from the command line using the <code>-OVF</code> and <code>-OVERLAY</code> options.

Scatter loading format is designed for embedded applications. It also supports overlays paged from ROM.

Overlay format is designed for applications running on systems containing a hard disk or similar peripheral device.

**Note**     *These two format options are mutually exclusive.*

The ARM linker allows the application image produced to be built to match the target system's memory map.

In the simple case, a memory map may have one ROM and one RAM area, which means the `-RO` and `-RW` options can be used to cause code and data to be mapped to the appropriate location (see **Chapter 13, Writing code for ROM** for details).

However, in embedded systems a more complicated memory map is often used and the linker has the scatter loading output format to provide this functionality.



## 14.2 Scatter Loading

armlink's scatter loading mechanism supports a flexible means of controlling your system's address map. Scatter loading enables you to partition a program image into several regions of code and data which are spread throughout the address map, in a disjointed manner. Each region is placed in a contiguous chunk of address space. The locations of such regions can differ between load time and run time.

The linker generates the symbols necessary to allow the regions to be loaded into memory at addresses other than their execution addresses. For example, initialized read-write data can be loaded into ROM, but it must be copied to RAM when the program is executing.

Scatter loading can also support the use of overlays (where several sections of code occupy the same memory space and only the required section is paged in at any particular time). For more information on using scatter loading with overlays, see **14.4 Overlays using Scatter Loading** on page 14-28 (you may find it helpful to first read **14.3 Overlays** on page 14-15).

### 14.2.1 Definitions

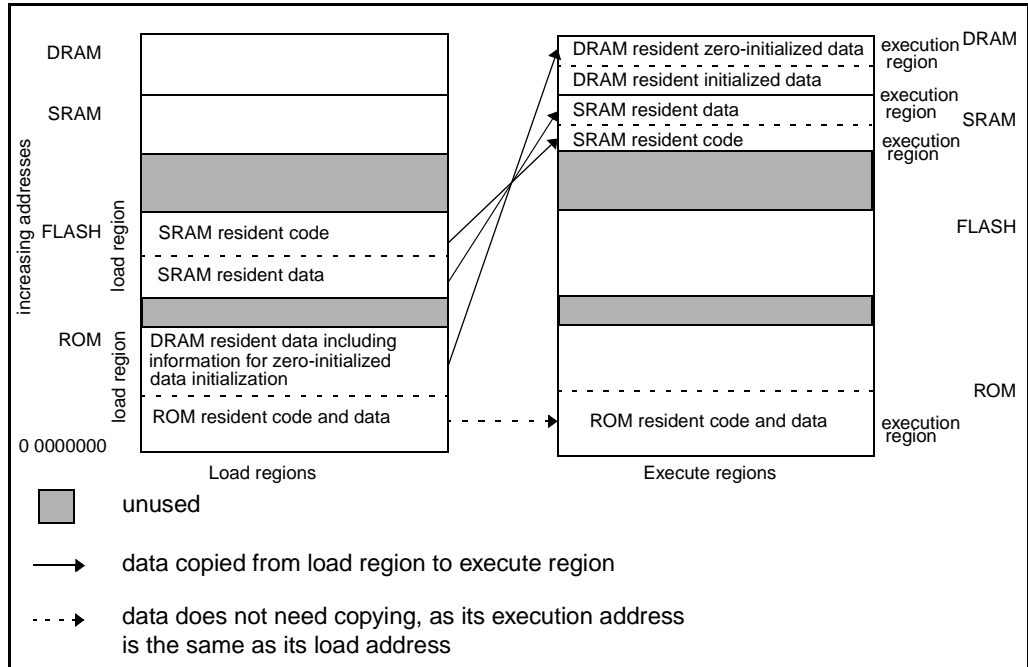
#### Load regions

The memory used by a program before it starts executing, but after it has been loaded into memory, can be split into a set of disjointed regions, each of which is a contiguous chunk of memory. These are called *load regions*.

#### Execute regions

The memory used by a program when it is executing can also be split into a set of disjointed regions. These are called *execute regions*.

# Placing Code and Data in Memory



**Figure 14-1: Simple example of scatter loading**

## 14.2.2 Scatter loading image formats

Scatter load images can be output in three formats:

`-bin -scf`

Generates a set of load region files into the directory given as the output filename. These can then be blown into ROM, Flash etc. as appropriate.

`-aif -bin -scf`

Generates a single extended AIF file suitable for loading into the debugger.

`-elf -scf`

Generates a single ELF file suitable for loading into the debugger.



# Placing Code and Data in Memory

## 14.2.3 Scatter loading description format

The details of a scatter loaded application are contained in a description file, the name of which is passed to the linker using the `-SCATTER` or `-SCOV` command line options.

The file format reflects the hierarchy of object areas, execute regions and load regions. An object area can be in precisely one execute region. An execute region can be in precisely one load region.

Lexically, a scatter load description is a sequence of tokens, whitespace and comments.

<b>special characters</b>	Single-characters with special significance are: <code>(){}", +</code> and <code>;</code> ( <code>LPAREN</code> , <code>RPAREN</code> , <code>LBRACE</code> , <code>RBRACE</code> , <code>QUOTE</code> , <code>COMMA</code> , <code>PLUS</code> and <code>SEMIC</code> ).
<b>tokens</b>	Tokens are <code>LPAREN</code> , <code>RPAREN</code> , <code>LBRACE</code> , <code>RBRACE</code> , <code>COMMA</code> , <code>PLUS</code> , <code>WORD</code> and <code>NUMBER</code> .
<b>comments</b>	A <code>SEMIC</code> following the end of a token begins a comment which extends to the end of the current line. This means that a <code>WORD</code> cannot begin with a <code>SEMIC</code> (unless it is enclosed in <code>QUOTES</code> ).
<b>numbers</b>	<p>A <code>NUMBER</code> has one of the forms:</p> <ul style="list-style-type: none"><li><code>"0"</code> octal-digit+</li><li><code>"&amp;"</code> hex-digit+</li><li><code>"0x"</code> hex-digit+</li><li><code>"0X"</code> hex-digit+</li><li>decimal-digit+</li></ul> <p>A <code>NUMBER</code> encodes a 32-bit unsigned value.</p>
<b>words</b>	A <code>WORD</code> is an alternation of quoted and unquoted <code>WORD</code> -segments.
<b>unquoted word segment</b>	An unquoted <code>WORD</code> -segment terminates on the first character in the set <code>{Whitespace, LPAREN, RPAREN, LBRACE, RBRACE, COMMA, PLUS, QUOTE}</code> .
<b>quoted word segment</b>	A quoted <code>WORD</code> -segment is enclosed by <code>QUOTE</code> characters and may contain any characters except <i>newline</i> . All other characters of which <i>isspace()</i> is true are translated to space. Two consecutive <code>QUOTES</code> stand for the literal <code>QUOTE</code> character and neither begin nor end a quoted <code>WORD</code> -segment.

Structurally, a scatter load description is a non-empty sequence of load region descriptions. For example:

```
Scatter-description ::= load-region-description+
```

# Placing Code and Data in Memory

---

## Load region description

A load region has a name, a base address, an optional maximum size and a non-empty list of execution regions. Formally:

```
load-region-description ::=  
    load-region-name base-address [ max-size ]  
    LBRACE execution-region-description+ RBRACE
```

An empty load region description has no obvious meaning and is forbidden. An empty execution region description (not to be confused with an empty execution region) is similarly meaningless.

Syntactically, *load-region-name* is a WORD. Only its first 31 characters are significant. In multi-file output formats (e.g. -BIN -SCATTER) *load-region-name* is used to name the file containing initialising data for this load region. On hosts which have shorter limits on directory entries, fewer characters are used.

The first 31 characters of *load-region-name* are also used by armlink to manufacture base and limit symbols (of the form Load\$\$name\$\$Base, Load\$\$name\$\$Limit) for the load region.

*base-address* is the address at which the contents of the region are loaded. It is a NUMBER (as described in the preceding section) and must be word-aligned, so &1234ABDC, 0x1e4,4000 and 0 are all acceptable, but 1234CD is not.

*max-size* is an optional NUMBER: if specified, the description is faulted if the region has more than this number of bytes allocated to it.

## Execute region description

An execution region is described by a name, a base address and an optional OVERLAY attribute. Formally:

```
execution-region-description ::=  
    exec-region-name base-address [ "OVERLAY" ]  
    LBRACE object-AREA-description+ RBRACE
```

Syntactically, *exec-region-name* is a WORD. Only its first 31 characters are significant. They are used by armlink to manufacture base and limit symbols (Image\$\$name\$\$base, etc.) describing the region and in armlink's diagnostic output.

*base-address* is the address at which objects in the region should be linked. It is a NUMBER and must be word-aligned.

armlink faults overlapping execution regions unless they have the "OVERLAY" attribute. For overlay regions that overlap, armlink builds clash maps and generates a reference to the overlay manager (which must already have been included in the image). Overlay segments are given names derived from the *exec-region-name*.

A non-OVERLAY execution region with its load address equal to its execution address is a *root* region. Only a root region may contain an entry point.

# Placing Code and Data in Memory

## Object area descriptions

An object-AREA-description is a pattern that identifies AREAs by:

- module name (object file name, library member name or library file name) and;
- AREA name or AREA attributes such as READ-ONLY, CODE, etc. Formally:

```
object-AREA-description ::=  
    module-selector-pattern [ LPAREN area-selectors RPAREN ]
```

An omitted `LPAREN area-selectors RPAREN` defaults to `(+RO)` (see below).

*area-selectors* is a comma-separated list of expressions. Each expression is a pattern against which the AREA name or the name of an attribute you want the selected AREA to have is matched. In the latter case the name must be preceded by a '+'. You may omit any comma immediately followed by a PLUS. Formally:

```
area-selectors ::=  
    (PLUS area-attrs | area-pattern )  
    ([ COMMA ] PLUS area-attrs | COMMA area-pattern )*
```

Additionally, the first occurrence of `FIRST` or `LAST` as an *area-attrs*, terminates the list.

Only AREAs that match both the *module-selector* and at least one *area-selector* are included in the execution region.

If an AREA matches more than one execution region, the matches are disambiguated as described below. If a unique match cannot be found, armlink faults the scatter description.

Note that the assignment of AREAs to regions is completely independent of the order in which patterns are written in the scatter load description.

## module\_selector patterns and area patterns

A *module-selector-pattern* and an *area-pattern* are patterns constructed from literal text, and the wildcard characters '\*' (matches 0 or more characters) and '?' (matches any single character). For example:

```
*armlib.* ; matches AREAs from any armlib.*
```

An AREA matches a *module-selector-pattern* if:

- the name of the object file containing the AREA or name of the library member (with no leading pathname) matches the *module-selector-pattern*
- the full name of the library from which the AREA was extracted matches the *module-selector-pattern*

Matching is case-insensitive, even on hosts with case-sensitive file naming.

# Placing Code and Data in Memory

---

## **area\_ selector**

An *area-selector* is:

- a pattern matched case-insensitively against the AREA's name
- an attribute selector matched against the area's attributes

An attribute selector follows a plus (+) character. The following selectors are recognized (case-insensitively):

RO-CODE  
RO-BASED-DATA  
RO-DATA (includes RO-BASED-DATA)  
RO (includes RO-CODE and RO-DATA)  
RW-CODE  
RW-BASED-DATA  
RW-STUB-DATA (shared library stub data)  
RW-DATA (includes RW-BASED-DATA and RW-STUB-DATA)  
RW (includes RW-CODE and RW-DATA)  
ZI  
ENTRY (the AREA containing the ENTRY point)

The following synonyms are also recognized:

CODE (= RO-CODE)  
CONST (= RO-DATA)  
TEXT (= RO)  
DATA (= RW)  
BSS (= ZI)

And also the pseudo attributes:

FIRST  
LAST

The pseudo-attributes `FIRST` and `LAST` can be used to mark the first and last AREAs in an execution region if the placement order is important (eg. if the `ENTRY` must be first and a checksum last).

Note that `RO-NOTBASED-DATA` cannot be specified directly. Rather, `RO-BASED-DATA` must be selected in one region and (less-specifically) `RO-DATA` in another.

# Placing Code and Data in Memory

## Disambiguating multiple matches

Every AREA is selected by a *module-selector* and an *area-selector*. Suppose AREA A matches  $\langle m1, s1 \rangle$  for execution region R1, and  $\langle m2, s2 \rangle$  for execution region R2. Then:

- assign A to R1 if  $\langle m1, s1 \rangle$  is more specific than  $\langle m2, s2 \rangle$
- assign A to R2 if  $\langle m2, s2 \rangle$  is more specific than  $\langle m1, s1 \rangle$
- otherwise diagnose the scatter description as faulty

Intuitively, a selector pattern  $p1$  is more specific than a pattern  $p2$  if every name that matches  $p1$  also matches  $p2$ , but not conversely. For example  $xyz$  is more specific than  $xyz^*$ .

A selector pair  $\langle m1, s1 \rangle$  is more specific than  $\langle m2, s2 \rangle$  if either:

- $m1$  is more specific than  $m2$ ; or
- $m2$  is not more specific than  $m1$  and  $s1$  is more specific than  $s2$

If  $s1$  and  $s2$  are both patterns matching AREA names, the same definition of  $s1$  more specific than  $s2$  holds as for  $m1$  more specific than  $m2$ .

If one of  $s1, s2$  matches an AREA name and the other matches AREA attributes, neither of  $s1$  and  $s2$  is more specific than the other.

If both  $s1$  and  $s2$  match AREA attributes then define  $s1$  more specific than  $s2$  ( $s1 < s2$ ) by:

```
RO-CODE < RO
RO-BASED-DATA < RO-DATA < RO
RW-CODE < RW
RW-BASED-DATA < RW-DATA < RW
RW-STUB-DATA < RW-DATA < RW
```

No other members of the  $s1 < s2$  relation between AREA attributes exist.

## Default root region specification

You can specify a default ROOT region to contain AREAs you do not assign to any other execution region. You can also specify a default ROOT-DATA region. If you do the unassigned read-only AREAs are placed in the default ROOT and unassigned read-write AREAs are placed in the default ROOT-DATA. You cannot specify a default ROOT-DATA region unless you specify a default ROOT.

You describe the default ROOT regions as follows:

```
ROOT root_load-address [ root_max_size ]
ROOT-DATA root_data_load_address
```

It is as if you wrote:

```
ROOT root_load_address {; the ROOT load region
    ROOT root_load_address { *(*) }
}
```

# Placing Code and Data in Memory

or

```
ROOT root_load_address { ; the ROOT load region
    ROOT root_load_address { *(+RO) }
    ROOT_DATA root_data_load_address { *(+RW,+ZI) }
}
```

as described in the subsections **Load region description** on page 14-6 and **Execute region description** on page 14-6.

The region names ROOT and ROOT-DATA are reserved for armlink.

If you do not specify a default ROOT, armlink faults any areas left unplaced by your scatter description.

### Example application

A particular application might use the following memory map:

Base	Size	Name
0x0	0x8000	ROM
0x8000	0x8000	SRAM
0x20000	0x40000	EEPROM
0x100000	0x100000	DRAM

Table 14-1: Example memory map

The application consists of five object files:

```
init.o obj1.o obj2.o obj3.o obj4.o
```

obj4 requires a large zero-initialized data area. This is to be placed in DRAM. The remaining read-write data areas are to be placed in SRAM along with the code from obj3.o. Object files obj2.o and init.o are to be placed in the root.

The description file for this application is:

```
ROOT 0x0 ; Specify the root region base address.
ROOT-DATA 0x8000
EEPROM 0x20000 0x40000 {
    EEPROM 0x20000 {
        obj1.o(+RO) ; Read Only AREAs to execute in the EEPROM.
        obj4.o(+RO) ; This region has the same execution address
                    ; as its load address.
    }
    SRAM 0x9000 {
        obj3.o
```



# Placing Code and Data in Memory

```
obj1.o(+RW,+ZI) obj4.o(+RW)
}
DATA 0x100000 { obj4.o(+ZI) }
}
```

## 14.2.4 Initialization

The linker does not provide the code that creates detailed execute regions from the load regions. Instead it generates sufficient information for a user-written routine to initialize all the execute regions that have base addresses not equal to their load addresses. This information is in the form of symbols specifying the length, execution address and load address of each region.

For zero-initialized data:

Image\$\$region_name\$\$ZI\$Base	execution address of the region
Image\$\$region_name\$\$ZI\$Length	execute region length in bytes (multiple of 4 bytes)

For ReadOnly and ReadWrite areas:

Load\$\$region_name\$Base	load address of the region
Image\$\$region_name\$Base	execution address of the region
Image\$\$region_name\$Length	execute region length in bytes (multiple of 4 bytes)

**Note** *A scatter load image is not padded out with zeros, but instead is always requires ZI data areas to be dynamically created. This is similar to the case with a normal -bin file when the -nozeropad option is used. There is therefore no need for a load address symbol for ZI data.*

The linker sorts AREAs within execute regions in the same order as for an AIF image. Hence non zero-initialized data that needs copying is contiguous.

A user could construct an initialization routine using these symbols. The root *Procedure Call Indirection Table (PCIT)* will be in the root read-write execute region and is included in the copying operation for the root data.

In the example, the initialization could be expressed in the following pseudo C:

```
void Init(void)
{
    __copy(Image$$root$Base, Load$$root$Base, Image$$root$Length);
    __copy(Image$$SRAM$Base, Load$$SRAM$Base, Image$$SRAM$Length);
    __zero(Image$$root$ZI$Base, Image$$root$ZI$Length);
    __zero(Image$$SRAM$ZI$Base, Image$$SRAM$ZI$Length);
    __zero(Image$$DRAM$ZI$Base, Image$$DRAM$ZI$Length);
}
```

where `__copy` performs a fast copy in the manner of `memcpy`, and `__zero` zeroes the specified number of bytes.

# Placing Code and Data in Memory

---

The initialization function needs to be called before the application main program is entered.

A simple implementation of an initialization routine is listed below. This uses two tables to control the initialization process. The first table lists the lengths and execution addresses of zero-initialized data. The second specifies the lengths, and the load and execution addresses of the execute regions that need to be copied. Both tables are terminated by a word containing zero.

```
        AREA InitApp, CODE, READONLY
        EXPORT initializeApp

initializeApp
        ADR     r0,ziTable
        MOV     r3,#0

ziLoop
        LDR     r1,[r0],#4
        CMP     r1,#0
        BEQ     initLoop
        LDR     r2,[r0],#4

ziFillLoop
        STR     r3,[r2],#4
        SUBS    r1,r1,#4
        BNE     ziFillLoop
        B       ziLoop

initLoop
        LDR     r1,[r0],#4
        CMP     r1,#0
        MOVEQ   pc,lr
        LDMIA   r0!,{r2,r3}
        CMP     r1,#16
        BLT     copyWords

copy4Words
        LDMIA   r3!,{r4,r5,r6,r7}
        STMIA   r2!,{r4,r5,r6,r7}
        SUBS    r1,r1,#16
        BGT     copy4Words
        BEQ     initLoop

copyWords
        SUBS    r1,r1,#8
        LDMIA   r3!,{r4,r5}
        STMIA   r2!,{r4,r5}
        BEQ     initLoop

        LDR     r4,[r3]
        STR     r4,[r2]

        B       initLoop

;
```



# Placing Code and Data in Memory

```
; A couple of MACROS to make the table entries easier to add.
; The execname parameter is the name of execution to initialize or
; copy.
;
        MACRO
        ZIEntry $execname
        LCLS    lensym
        LCLS    basesym
        LCLS    namecp
namecp SETS    "$execname"
lensym SETS    "| Image$$":CC:namecp:CC:"$$ZI$$Length| "
basesym SETS    "| Image$$":CC:namecp:CC:"$$ZI$$Base| "
        IMPORT $lensym
        IMPORT $basesym
        DCD    $lensym
        DCD    $basesym
        MEND

        MACRO
        InitEntry $execname
        LCLS    lensym
        LCLS    basesym
        LCLS    loadsym
        LCLS    namecp
namecp SETS    "$execname"
lensym SETS    "| Image$$":CC:namecp:CC:"$$Length| "
basesym SETS    "| Image$$":CC:namecp:CC:"$$Base| "
loadsym SETS    "| Load$$":CC:namecp:CC:"$$Base| "
        IMPORT $lensym
        IMPORT $basesym
        IMPORT $loadsym
        DCD    $lensym
        DCD    $basesym
        DCD    $loadsym
        MEND

ziTable
        ZIEntry root            ; Zero initialized data from the root read write
                                ; region
        DCD    0

InitTable
        InitEntry root          ; initialized data from the root read/write region
        DCD    0
        END
```

Each execute region that requires initialization of zero-initialized data must have an entry in `ziTable` of the form:

```
        ZIEntry    name
```

# Placing Code and Data in Memory

---

where *name* is the name of the execute region. Similarly, each execute region that needs to be copied must have an entry in `InitTable` of the form:

```
InitEntry name
```

The `initializeApp` routine is not called automatically at startup: it must be called explicitly before the application main program is entered.

This code can be found in the `initapp.s` file in directory `examples/scatter`.

## 14.3 Overlays

The linker supports both static and dynamic overlays by generating tables through which calls to overlay segments are redirected. If the relevant overlay segment is not loaded, a section of code, called the *overlay manager*, is called to load it. Although the linker generates references to the overlay manager, the linker does not provide it. An object file containing the overlay manager must be supplied in the link command. For a detailed description, see **14.3.5 The overlay manager** on page 14-19.

**Thumb** It is not possible to enter or exit an overlay segment in Thumb state. Hence all the external callable interfaces in an overlay segment should be ARM-state code.

The `-OVERLAY` linker option causes the linker to compute the size of the overlay segments automatically, and to output distinct memory partitions. The linker generates a set of files in a directory specified by the `-OUTPUT` option. Overlay segments are forced to specific memory addresses in a simple form of scatter loading. However, PCIT entries are generated even for non-clashing overlays, producing extra overheads in terms of code size and execution speed. For this reason the `-OVERLAY` option is not recommended for generating scatter loaded images, and `-SCATTER` should be used instead.

See **14.4 Overlays using Scatter Loading** on page 14-28 for information on using scatter loading with overlays.

### 14.3.1 Static overlays

In the static case, a simple two-dimensional overlay scheme is supported. There is one root segment, and as many memory partitions as specified by the user (for example, 1\_, 2\_, etc.). Within each partition, some number of overlay segments (for example, 1\_1, 1\_2, ...) share the same area of memory. You specify the contents of each overlay segment and the linker calculates the size of each partition, allowing sufficient space for the largest segment in it. All addresses are calculated at link time so statically overlaid programs are not relocatable. A hypothetical example of the memory map for a statically overlaid program might be:

2_1	2_2	2_3		high address
1_1	1_2	1_3	1_4	
root segment				low address

Segments 1\_1, 1\_2, 1\_3 and 1\_4 share the same area of memory. Only one of these segments can be loaded at any given instant; the remainder must be on backing store.

Similarly, segments 2\_1, 2\_2 and 2\_3 share the 2\_ area of memory, but are entirely separate from the 1\_ partition.

# Placing Code and Data in Memory

---

It is a current restriction that an overlay segment name is of the form *partition\_segment* and contains 10 or fewer characters. Note that there is no requirement for *partition* and *segment* to be numeric as shown in the example: any alphanumeric characters are acceptable.

## 14.3.2 Dynamic overlays

A dynamic or relocatable overlay scheme is obtained by specifying the `-Relocatable` command-line option. In this case:

- the root segment is a (load-time) relocatable AIF image
- each overlay segment is a plain binary image with relocation directives appended

When using relocatable overlays, it is expected that:

- the overlay manager allocates memory for a segment when it is first referenced
- a segment is unloaded, and the memory it occupies freed, by an explicit call to the overlay manager

Here, you give each overlay segment a simple name (no embedded underscore), and let the linker link each as if it were in its own partition (dynamically allocated by the overlay manager).

If a two-dimensional naming scheme is used, the linker generates segment clash tables (see below), and segments can be unloaded implicitly by the overlay manager when a clashing segment is loaded. In effect, this supports the classification of dynamic overlay segments into disjointed sets of *non co-resident* objects.

A dynamic overlay segment (including a root segment) is followed by a sequence of relocation directives. The sequence is terminated by a word containing `-1`. Each directive is a 28-bit byte offset of a word or instruction to be relocated, together with a flag nibble in the most significant 4 bits of the word. Flag nibbles have the following meanings:

- |   |  |
|---|--|
| 0 | relocate a word in the root segment by the difference between the address at which the root was loaded and the address at which it was linked  |
| 1 | relocate a word in an overlay segment by the address of the root   |
| 2 | relocate a word in an overlay segment by the address of the segment  |
| 3 | relocate a B or BL from an overlay segment to the root segment, by the difference (in words) between the segment's address and the root's address  |
| 7 | relocate a B or BL from the root segment to an overlay segment, by the difference (in words) between the root's address and the segment's address, (such relocation directives always refer to a PCIT entry in an overlay segment, which is used to initialize a PCIT section in the root when the overlay segment is loaded; see <b>14.3.5 The overlay manager</b> on page 14-19 for further explanation) |

# Placing Code and Data in Memory

## 14.3.3 Assignment of AREAs to overlay segments

The linker assigns *ARM Object Format (AOF)* AREAs to overlay segments under user control (see below). Usually, a compiler produces one code AREA and one data AREA for each source file (called `C$$code` and `C$$data` when generated by the C compiler). The C compiler option `-ZO` allows each separate function to be compiled into a separate code AREA, allowing finer control of the assignment of functions to overlay segments, (but at the cost of slightly enlarged code and enlarged object files). The user controls the overlay structure by describing the assignment of certain AREAs to overlay segments. For each remaining AREA in the link list, the linker acts as follows:

- if all references to the AREA are from the same overlay segment, the AREA is included in that segment
- otherwise, the AREA is included in the root segment.

This strategy can never make an overlaid program use more memory than if the linker put all remaining AREAs in the root, but it can sometimes make it smaller.

By default, only code AREAs are included in overlay segments. Data AREAs can be forcibly included, but it is the user's responsibility to ensure that doing so is meaningful and safe.

On disc, an overlaid program is organized as a directory containing a root image and a collection of overlay segments. The name of the directory is specified to the linker as the argument to its `-Output` flag.

The linker creates the following components within the application directory:

root segment	<code>root</code> , which is an AIF image
overlay segments	plain binary image fragments, for example:
	<code>1_1</code>
	<code>1_2</code>
	<code>...</code>
	<code>2_1</code>
	<code>...</code>

## 14.3.4 Describing an overlay structure to the linker

The overlay file, named as argument to the `-overlay` option, describes the required overlay structure. It is a sequence of *logical lines*:

- The backslash character (`\`) immediately before the end of a physical line continues the logical line on the next physical line.
- Any text from the semicolon (`;`) to the end of the logical line (inclusive) is a comment.

# Placing Code and Data in Memory

---

Each logical line has the following structure:

```
segment-name [ "(" base-address ")" ]  
module-name  [ "(" list-of-AREA-names ")" ]
```

where

<i>base-address</i>	is the address of the segment. The value can be expressed in decimal or hexadecimal. For example: 12 (decimal) 0x1FF0 (hexadecimal) &20C0 (hexadecimal)
<i>list-of-AREA-names</i>	is a comma-separated list. If omitted, all AREAs with the CODE attribute are included.
<i>module-name</i>	is either the name of an object file (with all leading pathname components removed), or the name of a library member (with all leading pathname components removed).

In the following example, `sort` would match the C library module of the same name:

```
1_1 edit1 edit2 editdata(C$$code,C$$data) sort
```

**Note** *These rules require that modules have unique names within a link list. For example, it is not possible to overlay a program made up from `test/thing.o` and `thing.o` (two modules called `thing`). This is a restriction on overlaid programs only.*

*To help partition a program between overlay segments, the linker can generate a list of inter-AREA references. This is requested by using the `-Xref` option. In general, if area A refers to area B, for example because `fx` in area A calls `fy` in area B, A and B should not share the same area of memory. Otherwise, every time `fx` calls `fy`, or `fy` returns to `fx`, there will be an overlay segment swap.*

*The `-MAP` option requests the linker to print the base address and size of every AREA in the output program. Although not restricted to use with overlaid programs, `-MAP` is most useful with them, as it shows how AREAs might be packed more efficiently into overlay segments.*

*Even though segments can be placed at specific memory locations by supplying base addresses, clash detection relies only on the names. For example, if `1_1` is placed at `0x8000`, and `1_2` is placed at `0x10000`, and `1_1` does not overlap `1_2`, the linker will still believe they clash, because they have the same partition name.*

## 14.3.5 The overlay manager

This section describes in detail how a static overlay manager operates. The details of a dynamic overlay manager are very similar. In both cases, details specific to the target system are omitted.

The job of the overlay manager is to load, swap, and unload overlay segments. This is done by trapping inter-segment function calls. References to data are resolved statically by the linker when each overlay segment is created. De-referencing a datum cannot cause an overlay segment to be loaded.

Every inter-segment procedure call is indirected through a table in the root segment that traps unloaded target overlay segments, PCIT. PCITs are created by the linker. Each overlay segment contains the data required to initialize its section of the PCIT when it is loaded. This is a table of Branch instructions, one for each function exported by the overlay segment. The linker knows the locations of each segment of the PCIT and of each function exported by each overlay segment, so it can create these Branch instructions at link time. In a dynamic overlay scheme, all segments, including the root, are assumed to be linked at 0, and a type 7 relocation directive is generated to describe the relocation of each of the initializing Branch instructions.

Initially, every sub-section of the PCIT in the root segment is initialized as follows (one for each procedure exported by the corresponding overlay segment):

```
STR LR, [PC, #-8]
```

A call to an entry in the root PCIT overwrites that entry, and every following entry, with the return address, until control falls off the end of that section of the PCIT into code that:

- finds which entry was called
- loads the corresponding overlay segment (and executes its relocation directives, if it is relocatable)
- overwrites the PCIT subsection with the associated branch vector (from the overlay segment that has just been loaded)
- retries the call

Future calls to this section of the PCIT will encounter instructions of the form `B ln`, adding only a few cycles to the procedure call overhead. This will persist until a function call, function return, or explicit call to the overlay manager causes this PCIT segment to be overwritten.

The load-segment code not only loads an overlay, but also re-initializes the PCIT sections corresponding to segments with which it cannot co-reside. It also installs handlers to catch returns to segments which have been unloaded.

The linker generates references to two symbols which must be defined by the overlay manager:

<code>Image\$\$Overlay_init</code>	initializes the overlay manager. This is done automatically when executing an AIF file.
<code>Image\$\$load_seg</code>	handles the loading of segments. This is called from unloaded segments' PCIT sections.

# Placing Code and Data in Memory

## The structure of a PCIT section

The per-PCIT-section code and data structures are shown below. These are created by the linker and used by the overlay manager. They are justified and explained in the following subsections. The space cost of this code is  $(9 + \#Clashes + \#Entries)$  words per overlay segment. Most of the work is done in the function `Image$$load_seg` (which is shared between all PCIT sections), and in `load_segment` (which is the common tail for both `Image$$load_seg` and `load_seg_and_ret`). For an explanation of `load_seg_and_ret`, see *Intercepting returns to overwritten segments* on page 14-23.

```
        STR        LR, [PC, #-8]; guard word
EntryV STR        LR, [PC, #-8]; > one entry for each
        ...                ; > procedure exported
        STR        LR, [PC, #-8]; > by this overlay segment
        BL         Image$$load_seg
PCITSection
VecsizeDCD    .-4-EntryV    ; size of entry vector
Base   DCD    ...          ; initialized by the linker
Limit  DCD    ...          ; initialized by the linker
Name   DCB    11 bytes     ; 10-char segment name + NUL
Flags  DCB    0            ; ...and a flag byte
ClashSzDCD    PCITEnd-.-4  ; size of table following
ClashesDCD    ...         ; >table of pointers or offsets
        ...         ; >to segments which cannot
        DCD    ...         ; >co-reside with this one
PCITEnd
```

Pointers to clashing segments point to the appropriate `PCITSection` labels (that is, into the middle of PCIT sections).

(If the overlays are relocatable, offsets between `PCITSection` labels are used rather than addresses which would themselves require relocation.)

We now define symbolic offsets from `PCITSection` for the data introduced here. These are used in the `Image$$load_seg` code described in the next subsection.

```
O_Vecsize    EQU Vecsize-PCITSection
O_Base       EQU Base-PCITSection
O_Limit      EQU Limit-PCITSection
O_Name       EQU Name-PCITSection
O_Flags      EQU Flags-PCITSection
O_ClashSz    EQU ClashSz-PCITSection
O_Clashes    EQU Clashes-PCITSection
```



# Placing Code and Data in Memory

## The Image\$\$load\_seg code

The Image\$\$load\_seg code contains a register save area which is shared with load\_seg\_and\_ret. Both of these code fragments are veneers on load\_segment. Both occur once in the overlay manager, not once per segment. Note that the register save area could be separated from the code and addressed via an address constant, as ip is available for use as a base register. Note also that load\_segment and its veneers preserve fp, sp, and sl, which is vital.

```
        STRLR    STR    LR, [PC, #-8]           ; a useful constant
        Rsave    %     10*4                     ; for r0-r8
        LRSave   %     4
        PCSave   %     4
Image$$load_seg
        STR      r8, RSave+9*4                   ; save a base register...
        ADR      r8, RSave
        STMIA    r8, {r0-r8}                     ; and some working registers
        MRS      r8,CPSR
        STR      r8,PSRSave
        LDR      r0,[LR,#-8]
        STR      r0, LRSave                      ; ...save it here ready for
                                                ; retry
        LDR      r0, STRLR                       ; look for this...
        SUB      r1, r8, #8                      ; ...starting at penultimate
                                                ; overwrite

01      LDR      r2, [r1, #-4]!
        CMP      r2, r0                          ; must stop on guard word...
        BNE      %B01
        ADD      r1, r1, #4                      ; gone one too far...
        STR      r1, PCSave                      ; where to resume
        B        load_segment                   ; ...and off to the common tail
```

On entry to load\_segment, r8 points to a register save for {r0-r8, LR, PC}, and r8 identifies the segment to be loaded. FP, SP and SL are preserved at all times by the overlay segment manager. There is only one copy of Image\$\$load\_seg, shared between all PCIT sections. A similar section of code, called load\_seg\_and\_ret, is invoked on return to an unloaded segment (see **Intercepting returns to overwritten segments** on page 14-23). This code is also a veneer on load\_segment which shares RSave, LRSave and PCSave, and which branches to load\_segment with r8 and r8 set up as described above.

**Note** *The code for STR LR, [PC, #-8] is 0xE50FE008. This address is unlikely to be in application code space, so overwriting indirection table entries with an application's return addresses is safe.*

# Placing Code and Data in Memory

## The load\_segment code

load\_segment must:

- re-initialize the global PCIT sections for any overlay segment that clashes with this one, while checking the stack for return addresses that are invalidated by so doing, and installing return handlers for them
- allocate memory for the about-to-be-loaded segment, if the overlay scheme is dynamic (this is system specific)
- load the required overlay segment (system specific)
- execute the loaded segment's relocation directives (if any)
- copy the overlay segment's PCIT into the global PCIT
- restore the saved register state (with `pc` and `lr` suitably modified)

On entry to `load_segment`, `r8` points to the register save area, and `r8` to the PCIT section of the segment to load. First the code must re-initialize the PCIT section (if any) that clashes with this one:

```
load_segment
    ADD    r1, r8, #O_Clashes
    LDR    r0, [r8, #O_ClashSz]
01  SUBS   r0, r0, #4
    BLT    Done_Reinit                ; nothing left to do
    LDR    r7, [r1], #4                ; a clashing segment...
    ADD    r7, r7, r8                 ; only if root is
                                      ; relocatable

    LDRB   r2, [r7, #O_Flags]
    CMPS   r2, #0                     ; is it loaded?
    BEQ    %B01                       ; no, so look again
    MOV    r0, #0
    STRB   r0, [r7, #O_Flags]         ; mark as unloaded
    LDR    r0, [r7, #O_Vecsize]
    SUB    r1, r7, #4                 ; end of vector
    LDR    r2, STRLR                  ; init value to store...
02  STR    r2, [r1, #-4]!              ;>
    SUBS   r0, r0, #4                 ;> loop to initialize the
                                      ; PCIT segment
    BGT    %B02                       ;>
```

Next, the stack of call frames for return addresses invalidated by loading this segment is checked, and handlers are installed for each invalidated return. This is discussed in detail in the next subsection. Note that `r8` identifies the segment being loaded, and `r7` the segment being unloaded.

```
BL check_for_invalidated_returns
```

Segment clashes have now been dealt with, as have the re-setting of the segment-loaded flags and the interception of invalidated returns. It is now time to load the required segment. This is system specific, so the details are omitted (the name of the segment is at offset `O_Name` from `r8`).

# Placing Code and Data in Memory

On return, calculate and store the real base and limit of the loaded segment and mark it as loaded:

```
BL      _host_load_segment           ; return base address in r0
LDR     r4, [r8, #PCITsect_Limit]
LDR     r1, [r8, #PCITsect_Base]
SUB     r1, r4, r1                   ; length
STR     r0, [r8, #PCITsect_Base]    ; real base
ADD     r0, r0, r1                   ; real limit
STR     r0, [r8, #PCITsect_Limit]
MOV     r1, #1
STRB    r1, [r8, #PCITsect_Flags]   ; loaded = 1
```

The segment's entry vector is at the end of the segment; it must be copied to the PCIT section identified by r8, and zeroed in case it is in use as zero-initialized data:

```
LDR     r1, [r8, #PCITsect_Vecsize]
ADD     r0, r0, r1                   ; end of loaded segment...
SUB     r3, r8, #8                   ; end of entry vector...
MOV     r4, #0                       ; for data initialization
01 LDR     r2, [r0, #-4]!              ;> loop to copy
STR     r4, [r0]                     ; (zero-init possible data
                                   ; section)
STR     r2, [r3], #-4                ;> the segment's PCIT
SUBS    r1, r1, #4                   ;> section into the
BGT     %B01                         ;> global PCIT...
```

Finally, continue execution:

```
LDR     r0, PSRSave
MSR     CPSR, r0
LDMIA   r8, {r0, r8, LR, PC}
```

## Intercepting returns to overwritten segments

The overlay scheme described so far is sufficient, provided no function call unloads any overlay in the current call chain. For example, consider a root segment and two procedures, A and B in overlays 1\_1 and 1\_2 respectively. Note that A and B may not be co-resident. In this situation, any pattern of calls like:

```
((root calls A, A returns)* (root calls B, B returns))*
```

works fine. Problems will occur if A calls B, because when B attempts to return, it will go to a random address within itself rather than to A.

# Placing Code and Data in Memory

To fix this deficiency, it is necessary to intercept some function returns. Trying to intercept all returns would be expensive; at the point of call there are no working registers available, and there is nowhere to store a return address, (the stack cannot be used without potentially destroying the current function call's arguments). The following observations describe an efficient implementation:

- a return address can only be invalidated by loading a segment which displaces a currently loaded segment
- at the point at which a segment is loaded, the stack contains a complete record of return addresses that might be invalidated by the load

Before loading a segment, check the procedure call backtrace (including the value stored in `LRSave`) for return addresses that fall in the segment about to be overwritten. Replace each such return address by a pointer to a return handler that loads the segment before continuing the return.

There is no simple way to avoid using a fixed pool of return handlers. You cannot use the stack (in a language-independent manner) because its layout is only partly defined during the function call. You could use a variant of the language-specific stack-extension code, but it would complicate the implementation significantly, and make some aspects of the overlay mechanism language specific. Similarly, it would be unwise to make any assumptions about the availability or management of heap space.

Using a fixed pool of handlers is not as bad as it first seems. A handler can only be needed if a call overwrites the calling segment. If this is done strictly non-recursively (meaning that if any `P` in segment 1 calls some `Q` in segment 2, then no `R` in segment 2 may call any `S` in segment 1 until `Q` has returned), the number of handlers required is bounded by the number of overlay segments. If recursive calls are made between overlay segments, performance will be very poor unless a large amount of work is done by each call.

**Note** *Only the most recent return should be allocated a return handler. For example, assume that there is a sequence of mutually recursive calls between segments A and B, followed by a call to C that unloads A. Only the latest return to A needs to be trapped, because as soon as A has been reloaded the remainder of the mutually recursive returns can unwind without being intercepted.*

### Return handler code

A return handler must store the real return address and the identity of the segment to return to (eg. the address of its PCIT section). It must also contain a call (indirectly) to the load segment code. In addition, it is assumed that the handler pool is managed as a singly-linked list. Then the handler code is:

```
BL      load_seg_and_ret
RealLR  DCD    0          ; space for the real return address
Segment DCD    0          ; -> PCIT section of segment to load
Link    DCD    0          ; -> next in stack order
```

`RealLR`, `Segment` and `Link` are set up by `check_for_invalidated_returns`.



# Placing Code and Data in Memory

## The load\_seg\_and\_ret code

HStack and HFree are set up by overlay\_mgr\_init, and maintained by check\_for\_invalidated\_returns. For simplicity, they are shown here as PC-relative-addressable variables. They are actually part of the data area shared with Image\$\$load\_seg. This data area can be addressed via an address constant, as ip is available as a base register.

```
HStack DCD0          ; top of stack of allocated handlers
HFree  DCD0          ; head of free-list

load_seg_and_ret
    STR    r8, RSave+9*4 ; save a base register...
    ADR    r8, RSave
    STMIA  r8, {r0-r8}   ; ... and some working registers
    MSR    r8,CPSR
    STR    r8,PSRSave
    LDMIA  LR,{r0,r1,r2}
    STR    r0, LRSave
    STR    0, PCSave
; Now unchain the handler and return it to the free pool
; (by hypothesis, HStack points to this handler...)
    STR    r2, HStack    ; new top of handler stack
    LDR    r2, HFree
    STR    r2, [r8, #8]   ; Link -> old HFree
    SUB    r2, r8, #4
    STR    r2, HFree      ; new free list
    MOV    r8, r1         ; segment to load
    B      load_segment
```

## The check\_for\_invalidated\_returns code

This code loads the segment identified by r8 into the slot identified by r7 to check LRSave and the chain of call-frames for the first invalidated return address. r7-r8, FP, SP and SL must be preserved.

```
    ADR    r6, LRSav      ; 1st location to check
    MOV    r0, FP         ; temporary FP...
01  LDR    r1, [r6]        ; the saved return address...
    LDR    r2, [r7, #0_Base]
    CMPS   r1, r2         ; see if >= base...
    BLT    %F02
    LDR    r2, [r7, #0_Limit]
    CMPS   r1, r2         ; ...and < limit
    BLT    FoundClash
02  CMPS   r0, #0         ; bottom of stack?
    MOVEQ  PC, LR         ; yes => return
```

# Placing Code and Data in Memory

---

```
SUB    r6, r0, #4
LDR    r0, [r0, #-12] ; previous FP
B      %B01
```

A handler is allocated for a segment containing a return address invalidated by the segment load:

```
FoundClash
    LDR    r0, HFree          ; head of chain of free handlers
    CMPS   r0, #0
    BEQ    NoHandlersLeft

                                ; transfer the next free handler
                                ; to head
                                ; of the handler stack.
    LDR    r1, [r0, #12]     ; next free handler
    STR    r1, HFree
    LDR    r1, HStack        ; the active handler stack
    STR    r1, [r0, #12]
    STR    r0, HStack        ; now with the latest handler linked
                                ; in, initialize the handler with a BL
                                ; load_seg_and_ret, RealLR and
                                ; Segment.
    ADR    r1, load_seg_and_ret
    SUB    r1, r1, r0        ; byte offset for BL in handler
    SUB    r1, r1, #8        ; correct for PC off by 8
    MOV    r1, r1, ASR #2; word offset
    BIC    r1, #&FF000000
    ORR    r1, #&EB000000    ; code for BL
    STR    r1, [r0]
    LDR    r1, [r6]
    STR    r6, [r0, #4]      ; RealLR
    STR    r0, [r6]          ; patch stack to return to handler
    STR    r7, [r0, #8]      ; segment to reload on return
    MOVS   PC, LR            ; and return

NoHandlersLeft
...                          ; initial creation of handler pool
                                ; omitted for brevity.
```

# Placing Code and Data in Memory

---

## Overlay manager code

When using the `-OVERLAY` option, the overlay manager code would be:

```
Retry
;
;      Call a routine to load the overlay segment.
;      First parameter is the length of the segment name.
;      The second parameter is the address of the segment name.
;      The third parameter is the base address of the segment.
;      The routine returns the segment length in r0.
;
      MOV     r0,#12
      ADD     r1, r8, #PCITsect_Name
      LDR     r2, [ r8, #PCITsect_Base]
      BL      LoadOverlaySegment

      TEQ     r0,#0
      MOVEQ   r0,#2
      BEQ     SevereErrorHandler
```

`LoadOverlaySegment` loads the named segment. In a non-embedded environment, this routine would load the segment from a file. This is the case in `overmgrs.s` in directory `examples/overlay`. In an embedded environment where the code is in some form of nonvolatile memory, the overlay segments would need to be packaged up with sufficient information for a `LoadOverlaySegment` implementation to load the segments correctly.

For example, the overlay could be put into a pseudo file system in nonvolatile memory and the segments accessed by name. This “packaging up” operation would need to be carried out after linking. The ARM Software Development Toolkit does not do this, as it will be highly specific to the application’s runtime environment. In the overlay example supplied with the toolkit, the overlay manager initialization routine is used to copy read/write data from the load address to the execution address.

# Placing Code and Data in Memory

---

## 14.4 Overlays using Scatter Loading

It is possible to specify overlapping execute regions in your scatter load description file, provided that they are specified as overlays. (If you do not specify overlapping execute regions as overlays, they are detected and flagged as errors.) There is also a restriction on overlapping overlays. The linker uses the same mechanism to detect clashing overlay regions as it does to detect clashing overlay segments in the overlay scheme, so the overlay region names are expected to be of the form:

`partition_segment`

Overlaid execute regions with the same partition name are deemed to clash by the linker. If the linker detects two overlaid execute regions that do not have the same partition name, the following error message is generated:

Overlaid regions *region1* and *region2* clash unexpectedly

where *region1* and *region2* are replaced by the names of the clashing execute regions.

A user can specify a size limit for a load region. If the size of a load region exceeds this value an error is flagged.

The user can position execute regions easily. However if the user wishes to have a set of execute regions continuous in memory when executing, there is no mechanism in the scatter loading description file to specify this.

All the overlay segments must have an execution address specified in the scatter load description file. The linker will not place overlay segments automatically. The scatter loading scheme does not support dynamic overlays. With scatter loading, PCIT information is not generated for execute regions not marked as overlays, so these regions do not have any overlay overhead associated with them.

The `-SCATTER` linker option instructs the linker to create either an extended AIF file or a directory of files. The overlays is placed into load regions and the linker adds information to the executable to allow the overlay manager to copy the overlay segments from the correct load region. The directory of output files are suitable for use in a ROM-based system.

### 14.4.1 Overlay handling

If overlays are present, the root PCIT is generated together with the information necessary to copy the PCIT to a read-write execute region (see **14.3.5 The overlay manager** on page 14-19 for a definition of PCIT).

A table of information that enables an overlay manager to perform overlay paging is generated. This table is referred to by a linker generated symbol: `Root$$OverlayTable`. The first word in the table is the number of entries in the table. The table is a sequence of entries each three words long. There is one entry per overlay segment.



# Placing Code and Data in Memory

Each entry contains:

Word 0	segment length in bytes
Word 1	execution address of the PCIT section for this overlay
Word 2	load address of the overlay

Overlays cannot be put into the root load region.

The linker generates references to the symbols `Image$$overlay_init` and `Image$$load_seg`. `Image$$load_seg` refers to the routine used to move segments to their execution addresses when *paged in*.

## 14.4.2 Segment clash detection

Clash detection relies on the *name* of an overlay segment rather than its base address and size. The linker attempts to find an underscore character (`_`) in the name, and on finding one takes the preceding string to be the partition name. Two segments are deemed to clash if they have the same partition names: for example, `seg_test` and `seg_eval` clash because the partition name is `seg` in both cases, while the names `alt_reset` and `pri_reset` do not.

**Note** *The overlay system's underlying mechanisms, such as the PCIT, rely on the processor executing in ARM state. Therefore Thumb-aware processors cannot call overlays while operating in Thumb state.*

## 14.4.3 The overlay manager

The overlay manager for scatter loaded overlays and the conventional overlay scheme are very similar. Indeed, only the segment loading code need be different. For a scatter loaded application, the code is of the form:

```
Retry
; Use the overlay table generated by the linker. The table format
; is as follows:
; The first word in the table contains the number of entries in
; the table.
; There follows that number of table entries. Each entry is 3 words
; long:
;     Word 1 Length of the segment in bytes.
;     Word 2 execution address of the PCIT section address. This is
;             compared against the value in r8. If the values are
;             equal we have found the entry for the called overlay.
;     Word 3 Load address of the segment.
; Segment names are not used.
;
IMPORT |Root$$OverlayTable|
LDR    r0,=|Root$$OverlayTable|
LDR    r1,[r0],#4
```

# Placing Code and Data in Memory

---

```
search_loop
    CMP     r1,#0
    MOVEQ   r0,#2      ; The end the table has been reached
    BEQ     SevereErrorHandler ;and the segment has not been found

    LDMIA   r0!,{r2,r3,r4}
    CMP     r8,r3
    SUBNE   r1,r1,#1
    BNE     search_loop

    LDR     r0,[ r8, #PCITsect_Base ]
    MOV     r1,r4
    MOV     r4,r2
    BL      MemCopy
```

where:

- `Root$$OverlayTable` is a symbol bound to the address of the linker-generated overlay information table.
- `SevereErrorHandler` is a routine called when the overlay manager detects an error.
- `MemCopy` is a system-specific memory copy routine where `r0` points to the destination area, `r1` points to the source area, and `r2` is the block size in bytes.

In the scatter loaded example supplied with the toolkit (in the `scatter.s` file in directory `examples/scatter`), the overlay manager initialization routine has no work to do, as all memory copying and initialization is done as part of the scatter loaded image initialization.

# 15

## Floating-Point Support

15.1	Introduction	15-2
15.2	The ARM Floating-point Library	15-3
15.3	Floating-Point Instructions	15-8
15.4	Linking the FPE into an Application	15-13
15.5	Configuring the FPA Support Code/FPE for a New Environment	15-14
15.6	Controlling Floating-Point Exceptions	15-15

# Floating-Point Support

---

## 15.1 Introduction

Floating-point arithmetic on the ARM can be done in four ways:

- by means of the software floating-point library
- by use of the (*Floating Point Emulator*) FPE or FPASC (FPA Support Code) in an existing environment
- by linking the FPE into the application
- by configuring the FPE or FPASC for a new environment and then using that environment

When the floating-point library is used, the compiler makes calls to the library routines to do floating-point calculations. For the other three options, the compiler instead uses floating-point instructions, which are either executed by the FPA and FPASC, or emulated by the FPE.

Use of the software floating-point library is the recommended option for most applications, because:

- it is about twice as fast as the FPE
- it usually results in smaller overall code size

The software floating-point library cannot, however, use a hardware FPA, and lacks some little-used facilities of the IEEE 754-1985 floating-point arithmetic standard. If either of these is required for a system, you are recommended to use one of the other options.

By default, `armcc` and `tcc` generate calls to floating-point library functions (see **15.2 The ARM Floating-point Library** on page 15-3). `armcc` can also be switched to generate floating-point coprocessor instructions (see **Chapter 10, Using the Procedure Call Standards**).

For more information on floating-point, refer to the *ARM FPA10 datasheet* (ARM DDI 0020I), and IEEE 754-1985 (the IEEE standard for binary floating-point arithmetic).

### 15.1.1 Thumb

The Thumb C compiler does not generate floating-point instructions, because these are not available in the Thumb instruction set. Therefore, the only direct solution is to use the software floating-point library.

### 15.1.2 Interworking

Because use of the floating-point library calls and use of floating-point instructions implies different procedure calling conventions, it is not possible to interwork between the two.

Interworking between the three options involving use of floating-point instructions is also not a good idea, because it increases code size (due to multiple copies of the FPE or FPASC being present), and in the case of the FPASC options, problems may result from the different copies of the FPASC competing for control of the FPA hardware.

## 15.2 The ARM Floating-point Library

The ARM software floating-point library provides a set of functions that the C compiler uses in place of the FPA instructions (eg. `_dadd` to add two doubles). Although based on the FPE, the library removes much of the overhead of emulating the FPA.

This approach has a number of advantages over the FPE:

- significantly faster code

By avoiding the decoding and emulation of the FPA instructions, the floating-point library typically achieves twice the floating-point performance of the FPE.

- smaller code

Although the executable for a given program is larger because all the floating-point code is linked to it, the memory used in a system is likely to be less because there is no need to include the FPE. For example, the `linpack` program increases in size from 24KB to 34KB, but no longer needs 26KB of FPE.

- no need to port the FPE to your target environment

The FPE must be modified for use in a new environment because it effectively forms part of the operating system. It requires some dedicated workspace that must be allocated in the target memory map. Multi-tasking environments must preserve the floating-point context between task switches. There is no need for any porting if the software library is used. This reduces development time significantly.

The main disadvantage is that code compiled using the library does not take advantage of a hardware floating-point accelerator. It also cannot make use of the following little-used facilities of the IEEE standard:

- underflow exceptions
- inexact exceptions
- trapped exceptions
- directed rounding modes
- extended precision

# Floating-Point Support

---

## 15.2.1 Usage

There is an APCS option to control which floating-point mechanism is used by armcc:

- apcs /softfp generates calls to ARM software floating-point library (default)
- apcs /hardfp generates in-line ARM floating-point instructions

Note that the compiler warns if you use `softfp` in conjunction with the following because these options only apply to a `hardfp` system:

- apcs /fpe3
- apcs /fpe2
- apcs /fpregargs

**Thumb** tcc always uses software floating-point.

## 15.2.2 Interworking between hardfp and softfp systems

Functions that return a floating-point type using the software floating-point library use the ARM's integer registers (returning a double in r0 and r1, and a float in r0). Under the FPE results are returned in the floating-point register f0. Hence the two are not compatible.

You should not need to mix ARM floating-point instructions and calls to the `softfp` library.

## 15.2.3 Calling the floating-point library from assembler

The software floating-point library provides a number of functions for basic floating-point operations. IEEE double-precision (`double`) values are passed in pairs of registers, and single-precision (`float`) numbers in a single register.

For example `_dadd` is the function to add two double-precision numbers and return the result.

It can be considered as having the prototype:

```
extern double _dadd(double, double);
```

that is, the two numbers to be added are passed in r0/r1 and r2/r3, and the result is returned in r0/r1.

Similarly the function `_fadd` (single-precision add) has the two arguments passed in r0 and r1, and the result returned in r0.

# Floating-Point Support

The complete set of functions are given in **Table 15-1: Library functions**

Function	Operation	Arg1 (type)	Arg2 (type)	Result (type)
<code>_dadd</code>	$A+B$	R0/R1 (double)	R2/R3 (double)	R0/R1 (double)
<code>_dsub</code>	$A-B$	R0/R1 (double)	R2/R3 (double)	R0/R1 (double)
<code>_drsb</code>	$B-A$	R0/R1 (double)	R2/R3 (double)	R0/R1 (double)
<code>_dmul</code>	$A*B$	R0/R1 (double)	R2/R3 (double)	R0/R1 (double)
<code>_ddiv</code>	$A/B$	R0/R1 (double)	R2/R3 (double)	R0/R1 (double)
<code>_drdv</code>	$B/A$	R0/R1 (double)	R2/R3 (double)	R0/R1 (double)
<code>_dneg</code>	$-A$	R0/R1 (double)		R0/R1 (double)
<code>_fadd</code>	$A+B$	R0 (float)	R1 (float)	R0 (float)
<code>_fsub</code>	$A-B$	R0 (float)	R1 (float)	R0 (float)
<code>_frsb</code>	$B-A$	R0 (float)	R1 (float)	R0 (float)
<code>_fmul</code>	$A*B$	R0 (float)	R1 (float)	R0 (float)
<code>_fdiv</code>	$A/B$	R0 (float)	R1 (float)	R0 (float)
<code>_frdv</code>	$B/A$	R0 (float)	R1 (float)	R0 (float)
<code>_fneg</code>	$-A$	R0 (float)		R0 (float)
<code>_dgr</code>	$A>B$	R0/R1 (double)	R2/R3 (double)	R0 (boolean)
<code>_dgeq</code>	$A\geq B$	R0/R1 (double)	R2/R3 (double)	R0 (boolean)
<code>_dls</code>	$A<B$	R0/R1 (double)	R2/R3 (double)	R0 (boolean)
<code>_dleq</code>	$A\leq B$	R0/R1 (double)	R2/R3 (double)	R0 (boolean)
<code>_dneq</code>	$A\neq B$	R0/R1 (double)	R2/R3 (double)	R0 (boolean)
<code>_deq</code>	$A==B$	R0/R1 (double)	R2/R3 (double)	R0 (boolean)
<code>_fgr</code>	$A>B$	R0 (float)	R1 (float)	R0 (boolean)
<code>_fgeq</code>	$A\geq B$	R0 (float)	R1 (float)	R0 (boolean)

**Table 15-1: Library functions**

# Floating-Point Support

Function	Operation	Arg1 (type)	Arg2 (type)	Result (type)
_fls	A<B	R0 (float)	R1 (float)	R0 (boolean)
_fleq	A<=B	R0 (float)	R1 (float)	R0 (boolean)
_fneq	A!=B	R0 (float)	R1 (float)	R0 (boolean)
_feq	A==B	R0 (float)	R1 (float)	R0 (boolean)
_dflt	(double)A	R0 (int)		R0/R1 (double)
_dfltu	(double)A	R0 (unsigned)		R0/R1 (double)
_dfix	(int)A	R0/R1 (double)		R0 (int)
_dfixu	(unsigned)A	R0/R1 (double)		R0 (unsigned)
_fflt	(float)A	R0 (int)		R0 (float)
_ffltu	(float)A	R0 (unsigned)		R0 (float)
_ffix	(int)A	R0 (float)		R0 (int)
_ffixu	(unsigned)A	R0 (float)		R0 (unsigned)
_d2f	(double)A	R0 (float)		R0/R1 (double)
_f2d	(float)A	R0/R1 (double)		R0 (float)

Table 15-1: Library functions (Continued)

**Thumb** In the Thumb software floating-point library, the Thumb entry points are predefined with `__16` (eg. `__16_dadd`). This is to allow both ARM and Thumb versions of floating-point functions to be present when interworking ARM and Thumb.





## 15.2.4 Formats library functions

int and unsigned	32-bit integer quantities.
boolean	either 0 (False) or 1 (True)
float	IEEE single precision floating-point number. See <b>Figure 15-1: IEEE single precision</b> .
double	IEEE double precision floating-point number. See <b>Figure 15-2: IEEE double precision</b> .

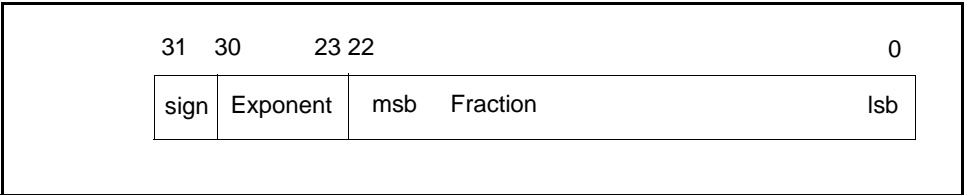


Figure 15-1: IEEE single precision

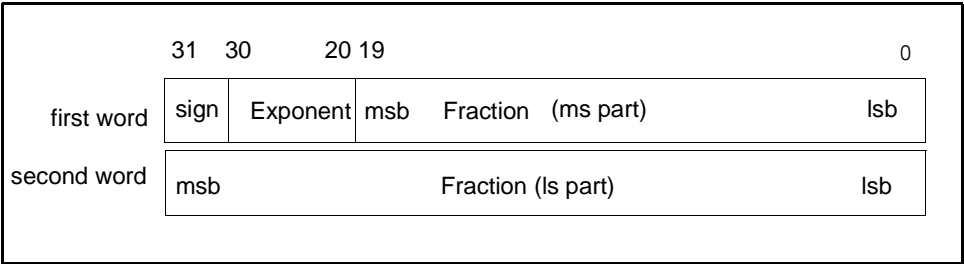


Figure 15-2: IEEE double precision

### Endianness

When stored, the first word of a double is always at the lower address in memory, and the second word is always at an address higher than the first word, regardless of the system's endianness.

# Floating-Point Support

## 15.3 Floating-Point Instructions

The ARM assembler supports a comprehensive floating-point instruction set. Whether implemented by hardware coprocessor or software emulation, floating-point operations are performed to the IEEE 754 standard. There are eight floating-point registers, numbered f0 to f7. Floating-point operations, like integer operations, are performed between registers.

**Note** *Floating-point operations are only usable from armcc and in ARM Assembly modules (not from Thumb code as this does not have the coprocessor instructions)*

Precision must be specified for many floating-point operations where shown as *prec* below. The options are:

S	single
D	double
E	extended
P	packed BCD (only available for LDF and STF instructions)

The rounding mode, shown below as *round*, defaults to round to nearest, but can optionally be set in the appropriate instructions to: *P* (round to +infinity), *M* (round to -infinity) or *Z* (round to zero).

In all the following instruction patterns, *Rx* represents an ARM register, and *Fx* a floating-point register.

### 15.3.1 Floating-point data transfer—LDF and STF

LDF	load data to floating-point register
STF	store data from floating-point register

The syntax of these instructions is:

```
op{condition}prec Fd,  [Rn,#offset]{!}  
                        [Rn]{, #offset}  
                        program-or-register-relative-expression
```

The memory address can be expressed in one of three ways, as shown above. In the first, pre-indexed form, an ARM register *Rn* holds the base address, to which an offset can be added if necessary. Writeback of the effective address to *Rn* can be enabled using *!* The offset must be divisible by 4, and within the range -1020 to 1020 bytes. With the second, post-indexed form, writeback of *Rn+offset* to *Rn* after the transfer is automatic, and the data is transformed from address *Rn*, not address *Rn* plus offset. Alternatively, a program-or-register-relative expression can be used, in which case the assembler generates a PC-or-register-relative, pre-indexed address; if it is out of range an error results.

## 15.3.2 Floating-point register transfer—FLT and FIX

FLT                      integer to floating-point transfer                       $F_n := R_d$

The syntax of this instruction is:

$FLT\{condition\}prec\{round\}F_n, R_d$

where  $R_d$  is an ARM register.

FIX                      floating-point to integer transfer                       $R_d := F_n$

The syntax of this instruction is:

$FIX\{condition\}\{round\} R_d, F_n$

## 15.3.3 Floating-point register transfer—status and control

The following instructions transfer values between the floating-point coprocessor's status and control registers, and an ARM general purpose register:

WFS                      write floating-point status                       $FPSR := R_d$

RFS                      read floating-point status                       $R_d := FPSR$

WFC                      write floating-point control                       $FPC := R_d$  (privileged modes only)

RFC                      read floating-point control                       $R_d := FPC$  (privileged modes only)

The syntax of the above four instructions is:

$opcode\{condition\} R_d$

WFC and RFC should never be used by code outside the floating-point system (that is, the FPA and FPASC or the FPE). They are only documented here for completeness.

## 15.3.4 Floating-point multiple data transfer—LFM and SFM

Note that these instructions are not supported by some older versions of the FPE.

LFM                      load floating-point multiple

SFM                      store floating-point multiple

These instructions are used for block data transfers between the floating-point registers and memory. Values are transferred in an internal 96-bit format, with no loss of precision and with no possibility of an IEEE exception occurring, (unlike STFE which may fault on storing a trapping NaN). There are two forms, depending on whether the instruction is being used for stacking operations or not. The first, nonstacking, form is:

$op\{condition\} F_d, count, [R_n]$   
 $[R_n, \#offset]\{!\}$   
 $[R_n], \#offset$

The first register to transfer is  $F_d$ , and the number of registers to transfer is  $count$ . Up to four registers can be transferred, always in ascending order. The count wraps round at f7, so if f6 is specified with four registers to transfer, f6, f7, f0 and f1 will be transferred in that order. With pre-indexed addressing, the destination/source register can be specified with or

# Floating-Point Support

without an *offset* expressed in bytes; writeback of the effective address to *Rn* can be specified with *!*. With post-indexed addressing (the third form above), writeback is automatically enabled, and the data is transferred from address *Rn*, not '*Rn* plus offset'. Note that *r15* cannot be used with writeback, and that *offset* must be divisible by 4 and in the range  $-1020$  to  $1020$ , as for other coprocessor loads and stores.

The second form adds a two-letter stacking mnemonic (below *ss*) to the instruction and optional condition codes. The mnemonic *FD* denotes a full, descending stack (pre-decrement push, post-increment pop), while *EA* denotes an empty, ascending stack (post-increment push, pre-decrement pop). The syntax is as follows:

```
opcode {condition}ss Fd,count,[Rn]{!}
```

*FD* and *EA* define pre- and post-indexing, and the up/down bit by reference to the form of stack required. Unlike the integer block-data transfer operations, only *FD* and *EA* stacks are supported. The character *!*, if present, enables writeback of the updated base address to *Rn*; *r15* cannot be the base register if writeback is enabled.

The possible combinations of mnemonics are listed below:

LFMFD	load floating-point multiple from a full stack, descending (post-increment load)
LFMEA	load floating-point multiple from an empty stack, ascending (pre-decrement load)
SFMFD	store floating-point multiple to a full stack, descending (pre-decrement store)
SFMEA	store floating-point multiple to an empty stack, ascending (post-increment store)

## 15.3.5 Floating-point comparisons—CMF and CNF

CMF	compare floating-point	compare <i>Fn</i> with <i>Fm</i>
CMFE		
CNF	compare negated floating-point	compare <i>Fn</i> with $-Fm$
CNFE		

The syntax of these instructions is:

```
opcode{condition} Fn,Fm
```

*CMF* and *CNF* only raise exceptions for signalling NaN operands and should be used to test for equality (*Z* clear/set) and unorderedness (*V* set/clear). To comply with IEEE 754-1985, all other tests should use *CMFE* or *CNFE*, which may raise an exception if either of the operands is any sort of NaN.



## 15.3.6 Floating-point binary operations

ADF	add	$Fd := Fn + Fm$
MUF	multiply	$Fd := Fn * Fm$
SUF	subtract	$Fd := Fn - Fm$
RSF	reverse subtract	$Fd := Fm - Fn$
DVF	divide	$Fd := Fn / Fm$
RDF	reverse divide	$Fd := Fm / Fn$
POW	power	$Fd := Fn$ to the power of $Fm$
RPW	reverse power	$Fd := Fm$ to the power of $Fn$
RMF	remainder	$Fd :=$ remainder of $Fn / Fm$
FML	fast multiply	$Fd := Fn * Fm$
FDV	fast divide	$Fd := Fn / Fm$
FRD	fast reverse divide	$Fd := Fm / Fn$
POL	polar angle	$Fd :=$ polar angle of $Fn, Fm$ ( $= \text{ATN}(Fm / Fn)$ whenever the quotient exists)

The syntax of these instructions is:

*binop*{*condition*}*prec*{*round*} *Fd*, *Fn*, *Fm*

*Fm* can be either a floating-point register, or one of the floating-point constants #0, #1, #2, #3, #4, #5, #10 or #0.5. Fast operations produce results that may only be accurate to single precision.

# Floating-Point Support

## 15.3.7 Floating-point unary operations

MVF	move	$Fd := Fm$
MNF	move negated	$Fd := -Fm$
ABS	absolute value	$Fd := \text{ABS}(Fm)$
RND	round to integral value	$Fd := \text{integer value of } Fm$ (using current rounding mode)
URD	unnormalized round	$Fd := \text{integer value of } Fm,$ possibly in abnormal form
NRM	normalize	$Fd := \text{normalised form of } Fm$
SQT	square root	$Fd := \text{square root of } Fm$
LOG	logarithm to base 10	$Fd := \log Fm$
LGN	logarithm to base e	$Fd := \ln Fm$
EXP	exponent	$Fd := e^{Fm}$
SIN	sine	$Fd := \text{sine of } Fm$
COS	cosine	$Fd := \text{cosine of } Fm$
TAN	tangent	$Fd := \text{tangent of } Fm$
ASN	arc sine	$Fd := \text{arc sine of } Fm$
ACS	arc cosine	$Fd := \text{arc cosine of } Fm$
ATN	arc tangent	$Fd := \text{arc tangent of } Fm$

The syntax of these instructions is:

$$unop\{condition\}prec\{round\} \; Fd, Fm$$

$Fm$  can be either a floating-point register or one of the floating-point constants #0, #1, #2, #3, #4, #5, #10 or #0.5.



## 15.4 Linking the FPE into an Application

FPE is supplied with the ARM C system as a linkable object file. Its environmental dependencies are all via a stub, supplied as an assembly language source. This stub file, `fpestub.s`, documents how to attach an FPE to the invalid instruction trap location (address 0x4).

The FPE and `fpestub` are linked together with whatever else is required to make a standalone module on the target hardware. The `fpestub.s` contains two entries for:

- |                |   |
|----------------|---|
| initialization | attaching it to the invalid instruction trap vector. This is called on activation of the standalone module.           |
| finalization   | removing it from the invalid instruction trap vector. This should be called on deactivation of the standalone module. |

For testing purposes, the FPE, `fpestub.s` and a test application can be linked together to make a single, standalone application. The application must call `__fp_initialise` before using any floating-point instructions, and `__fp_finalise` before exiting.

# Floating-Point Support

---

## 15.5 Configuring the FPA Support Code/FPE for a New Environment

For information on how to configure the FPASC and/or FPE for a new environment, see *Application Note 10 Configuring the FPA Support Code/FPE* (ARM DAI 0040).



## 15.6 Controlling Floating-Point Exceptions

Both the `hardfp` and `softfp` modes provide a function (`__fp_status`) for setting and reading the status of either the FPE/FPA or the floating-point library.

The following is an extract from `stdlib.h`:

```
extern unsigned int __fp_status(unsigned int /* mask */,
                               unsigned int /* flags */);
```

```
#define __fpsr_IXE 0x100000    inexact exception trap enable bit
#define __fpsr_UFE 0x80000     underflow exception trap enable bit
#define __fpsr_OFE 0x40000     overflow exception trap enable bit
#define __fpsr_DZE 0x20000     divide by zero exception trap enable bit
#define __fpsr_IOE 0x10000     invalid operation exception trap enable bit
#define __fpsr_IXC 0x10        inexact exception flag bit
#define __fpsr_UFC 0x8         underflow exception flag bit
#define __fpsr_OFc 0x4         overflow exception flag bit
#define __fpsr_DZC 0x2         divide by zero exception flag bit
#define __fpsr_IOC 0x1         invalid operation exception flag bit
```

`mask` and `flags` are bit fields that correspond directly to the floating-point status register (FPSR) in the FPE/FPA and the software floating-point library.

The function `__fp_status` returns the current value of the status register, and also sets the writable bits of the word (the exception control and flag bytes) to:

```
new = (old & ~mask) ^ flags;
```

Four different operations can be performed on each status register bit, determined by the respective bits in `mask` and `flags`:

mask bit	flags bit	effect
0	0	no effect
0	1	toggle bit in status register
1	0	clear bit in status register
1	1	set bit in status register

**Table 15-2: Status register bit operations**

The `__fp_status` function always returns the current value of the status register, before any changes are applied.

# Floating-Point Support

---

Initially all exceptions are enabled, and no flags are set.

## Examples

```
status = __fp_status(0,0);
/* reads the status register, does not change it */
__fp_status(__fpsr_DZE,0);
/* disable divide-by-zero exception trap */
inexact = __fp_status(__fpsr_OFC,0) & __fpsr_OFC;
/* read (and clear) overflow exception flag bit */
/* Report the type of floating-point system being used. */
switch (flags=(__fp_status(0,0)>>24))
{
    case 0x0: case 0x1:
        printf("Software emulation\n");
        break;
    case 0x40:
        printf("Software library\n");
        break;
    case 0x80: case 0x81:
        printf("Hardware\n");
        break;
    default:
        printf("Unknown ");
        if (flags & (1<<7))
            printf("hardware\n");
        else
            printf("software %s\n",
                flags & (1<<6) ? "library" : "emulation");
        break;
}
```

## System ID byte

Bits 31:24 contain a system ID byte. The currently defined values are:

0x00	pre-FPA floating-point emulator
0x01	FPA compatible floating-point emulator
0x40	floating-point library
0x80	FPPC (obsolete)
0x81	FPA10 (with FPSC module)

The top bit (bit 31) is used to distinguish between hardware and software systems, and bit 30 is used to distinguish between software emulators and libraries.

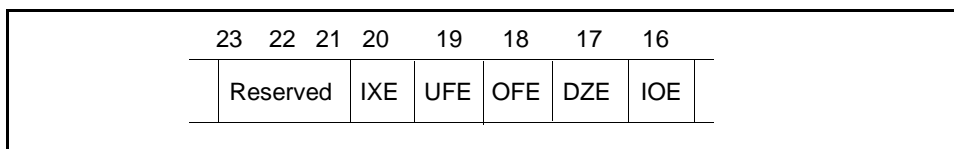
## Exception trap enable byte

Each bit of the exception trap enable byte corresponds to one type of floating-point exception.

Bits 23:16 control the enabling of exceptions on floating-point errors:

bits 23:21		reserved
bit 20	IXE	inexact exception enable*
bit 19	UFE	underflow exception enable*
bit 18	OFE	overflow exception enable
bit 17	DZE	divide by zero exception enable
bit 16	IOE	invalid operation exception enable

A set bit causes the system to take an exception trap if an error occurs. Otherwise a bit is set in the cumulative exception flags (see **Figure 15-3: Exception trap enable byte**) and the IEEE defined result is returned.



**Figure 15-3: Exception trap enable byte**

**Note** *The current floating-point library will never produce those exceptions marked with a \*. A bit in the cumulative exception flags byte is set as a result of executing a floating-point instruction only if the corresponding bit is not set in the exception trap enable byte. If the corresponding bit in the exception trap enable byte is set, a runtime error occurs (SIGFPE is raised in a C environment).*

## System control byte

This byte is not used on the floating-point library system. Refer to the FPA datasheet for details of its meaning under FPA and FPE systems.

In particular, the NaN exception control bit (bit 9) is not yet supported by the floating-point library, but may be in a future version.

## Exception flags byte

Bits 7:0 contain flags for whether each exception has occurred in the same order as the exception trap enable byte, see **Figure 15-4: Cumulative exception flags byte**. Exceptions occur as defined by IEEE 754.

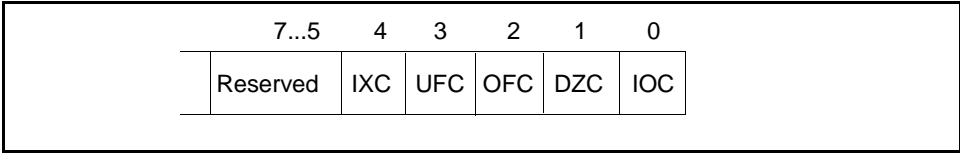


Figure 15-4: Cumulative exception flags byte

# Index

## Symbols

- \$ characters
  - in variable names 13-26
- \$semihosting\_vector 7-19
- \_\_main
  - undefined 13-25
- \_\_rt\_stkovf\_split\_big
  - undefined symbol 13-24

## Numerics

- 26-bit ARMs
  - Angel and Demon 6-30
- 64-bit
  - integer addition 10-5
  - multiplication result 10-14

## A

- adding of 64-bit integers 10-5
- addresses
  - loading into registers 9-19
  - memory 3-27
- ADP over JTAG 6-28
- ADP over JTAG using EmbeddedICE board 6-28
- ADR 9-22
- ADRL 9-22
- ADW *See* ARM Debugger for Windows
- ALU status flags 9-14
- Angel 3-4, 6-2
  - 26-bit ARMs 6-30
  - ADP over JTAG 6-28
  - and Demon 6-3, 6-29
  - ARM PID7T board 6-23
  - ARM60 PIE card 6-26
  - ARMulator 5-9
  - big- and little-endian operation 6-26

- channel viewers 3-31, 3-45
- channels 6-13
  - adding 6-22
- configuring 6-12
- configuring in ADW 3-31
- device drivers 6-14
- downloading debug agent 6-21
- EmbeddedICE 6-21, 6-27, 7-19
- exception vectors 6-12
- flash downloading 6-6, 6-24
- full and minimal 6-7
- Late Debugger Startup 6-14
- minimal 6-17
- overview of development process 6-4
- PID board 6-23
- porting 6-18
- ROM locations 6-20
- APCS
  - defined 10-3
  - floating point calls 15-4
  - inter-link-unit 10-4
  - register usage 10-8
  - stack chunk 10-4
  - static base 10-4
- APM *See* ARM Project Manager
- applications
  - entered at base address 13-3
  - entered via reset vector 13-3
- AREA
  - directive 9-5
  - overlay segments 14-17
- arguments
  - specifying while debugging 3-43
- ARM applications
  - debugging 3-2, 3-3
- ARM code 3-6
- ARM core 12-18
  - instruction set 9-3
  - veneers 12-10
- ARM Debugger for Windows 3-2
  - configuring 3-29
  - configuring EmbeddedICE 3-33
  - configuring target environments 3-28
  - desktop 3-11
  - exiting 3-27
  - getting started 3-21
  - new features 1-6
  - starting from APM 3-21
  - windows 3-12
- ARM PID
  - flash download 3-45
- ARM PID7T board 6-23
  - reset 7-25
- ARM PIE7 board
  - reset 7-25
- ARM Project Manager 2-2
  - build log 2-10
  - changing display 2-10
  - configuring 2-26
  - desktop 2-9
  - Edit Window 2-11
  - editor preferences 2-27
  - exiting 2-19
  - getting started 2-13
  - interworking ARM and Thumb 12-21
  - new features 1-5
  - Project Window 2-9
  - View Window 2-12
- ARM Software Development Toolkit
  - new features 1-5
- ARM state
  - and Thumb state 12-2
- ARM60 PIE card
  - Angel 6-26
- armasm 1-3
- armcc 1-3
  - c option 4-4
  - command-line example 4-2
  - generating assembly language 4-5
  - S option 4-5
- armlib 2-31
  - ARM Project Manager 2-12, 2-32
- armlink 1-3, 4-4
  - assignment of AREAs to overlay segments 14-17
  - describing an overlay structure to the linker 14-17
  - return handler code 14-24
  - segments 14-2
- armprof 8-19

- collecting data with ADW 3-8
- options 8-21
- armsd 1-3, 4-7
  - command-line example 4-3
  - configuring for EmbeddedICE 7-9
  - help 4-7
  - map files 8-8
- armsd.map 8-8
- armsd.map file 5-8
- armul.cnf 5-6
- ARMulator 1-3, 3-3, 5-2
  - Angel model 5-9
  - armsd.map file 5-8
  - armul.cnf file 5-6
  - can't go error 13-27
  - configuring in ADW 3-30
  - controlling using debugger 5-6
  - Demon 5-9
  - dummy MMU 5-11
  - models 5-2
  - profiler 5-12
  - profiling and instruction tracing 8-25
  - real time simulation 8-8
  - rebuilding 5-4
  - stubs 5-3
  - tracer 5-13
  - watchpoints 5-15
  - Windows Hourglass 5-15
- arrays
  - displayed in ADW 3-37
- assembler 9-2
  - ARM and Thumb 12-18
  - calling from C 9-25
  - combining with C 10-2
  - creating from C 4-5
  - floating-point calls 15-4
  - module 9-5
  - subroutines 9-7
- assembler module
  - Thumb 9-8

## B

- backtrace 3-6

- Backtrace Window 3-15
- benchmarking 8-6
- big- and little-endian 6-26
- binaries
  - viewing with APM 2-12, 2-31
- breakpoints 3-6
  - and EmbeddedICE 7-13
  - armsd 4-7
  - complex 3-39
  - EmbeddedICE vector 7-15
  - low-level symbols 3-40
  - removing 3-25
  - simple 3-24
- Breakpoints Window 3-15
- build
  - force build 2-5, 2-22
- build log 2-10
- build step patterns 2-7
  - adding 2-44
  - editing 2-43
  - overview 2-41
- build steps 2-5
  - assigning tools 2-42
  - performing single 2-30
- build tools
  - assigning to a build step 2-42
  - configuring 2-20
  - resetting configuration 2-21
- building 2-4
  - correcting problems 2-17
  - projects 2-16
  - stopping a build 2-25
  - variants 2-23

## C

- C
  - calling assembler 9-25
  - combining with assembler 10-2
  - using libraries in ROM 13-16
- c option
  - armcc 4-4
- call graph profiling 8-20
- channel viewers 3-31

- activating 3-45
- clash detection
  - overlays 14-29
- clock speeds 8-11
- clock()
  - EmbeddedICE 7-32
- code size
  - Dhrystone example 8-4
  - measuring 8-3
  - reducing 8-14
  - shorts 8-16
- code speed
  - setjmp() 8-18
- command line
  - compile, link, run 4-2
- Command Window 3-13
- command-line arguments
  - specifying while debugging 3-43
- command-line debugger
  - armsd 4-7
  - within ADW 3-44
- command-line pattern 2-42
- compile and link
  - separating 4-4
- complex
  - breakpoints 3-39
  - watchpoints 3-40
- conditional execution 9-14
- configuring
  - ARM Debugger for Windows 3-29
  - ARM Project Manager 2-26
  - build tools 2-20
  - resetting tools 2-21
  - target environment 3-28
- configuring in ADW
  - ARMulator 3-30
  - EmbeddedICE 3-33
  - Remote\_A 3-31
  - Remote\_D 3-32
- Console Window 3-14
- constants
  - loading 9-10
- converting projects 2-25
- creating
  - new projects 2-13

- source files 2-14, 2-28
- templates 2-39
- cursor
  - running execution to 3-23
- cycle counts
  - Dhrystone example 8-6
  - displaying 8-6

## D

- Data 11-28
- data abort 11-7
  - exception 11-2
  - handler 11-28
- debug agent 3-7
  - downloading 6-21
- debug comms channel
  - Angel 6-27
- debug extensions
  - EmbeddedICE 7-2
- debug monitor
  - See Angel 3-4
  - See Demon 3-5
- Debugger
  - ADW 3-2
  - command-line (armsd) 4-7
  - modifying variables (armsd) 4-10
  - variables 3-15
- Debugger Internals Window 3-15
- debugging
  - images from APM 2-19
  - ROM systems 7-24
  - systems 3-3
- decaof 1-3, 2-31
  - ARM Project Manager 2-12, 2-32
- decaxf 1-3, 2-31
  - ARM Project Manager 2-12, 2-32
- Demon 3-5
  - and Angel 6-3, 6-29
  - ARMulator 5-9
  - configuring in ADW 3-32
  - moving to EmbeddedICE 7-19
- description
  - editing project template 2-40



- description file
  - scatter loading 14-5
- details
  - editing project template 2-40
- development process 6-4
- Dhrystone
  - code size 8-4
  - example 8-4
  - map files 8-12
  - modifying for card specific timer 7-33
- directives
  - ABS 15-12
- disassembled code 3-7
  - displaying 3-37
- disassembly mode
  - specifying 3-38
- Disassembly Window 3-17
- display formats 3-36
  - changing 3-36
  - restoring default 3-36
- displaying
  - binaries 2-31
  - build messages 2-10
  - debugger variables (armsd) 4-10
  - disassembled code 3-37
  - projects 2-15
  - source code (armsd) 4-10
  - source files 3-34
  - variables (ADW) 3-15
- downloading debug agent 6-21
- dynamic overlays 14-16

## E

- Edit Window 2-11
- editing
  - complex breakpoints 3-39
  - complex watchpoints 3-40
  - registers 3-26
  - variables 3-26
- editor
  - Edit Window 2-11
  - preferences 2-27
- embedded C library 13-16

- EmbeddedICE 3-4, 7-2
  - accessing macrocell 7-28
  - adding SWI handlers 7-18
  - ADP over JTAG 6-28
  - Angel 6-21, 6-27, 7-19
  - breakpoints 7-13
  - configuring 7-7
  - configuring in ADW 3-33
  - connecting and powering up 7-6
  - debugging ROM systems 7-24
  - Demon 7-19
  - effect on debuggee 7-5
  - endianness 7-10
  - exceptions 7-15
  - floating-point coprocessor 7-39
  - interface 7-3
  - macrocell 7-3
  - reset and JTAG signal connection 7-20
  - reset, faking 7-24
  - timer accuracy 7-32
  - vector breakpoints 7-15
  - watchpoints 7-13
- enabling 3-31
- END
  - directive 9-6
- endianness
  - Angel 6-26
- ENTRY
  - directive 9-6
- entry point
  - not defined 13-26
- environment
  - configuring 3-28, 3-33
  - configuring ARMulator 3-30
  - configuring Remote\_A (Angel) 3-31
  - Remote\_D (Demon) 3-32
- errors
  - finding 2-17
- examining
  - memory 3-27
  - registers 3-26
  - variables 3-26
- exception handlers
  - data abort handler 11-28
  - installing 11-8

- interrupt 11-19
    - prefetch abort handler 11-27
    - reset handler 11-25
    - returning from 11-5
    - SWI handler 11-12
    - Thumb state 11-31
    - Thumb-aware processors 11-31
    - undefined instruction handler 11-26
  - exceptions 11-2
    - controlling floating-point 15-15
    - EmbeddedICE 7-15
    - entering 11-5
    - leaving 11-5
    - prefetch abort 11-2
    - priorities 11-3
    - response by processors 11-5
    - returning from 11-32
    - software interrupt 11-2
    - use of modes 11-3
    - use of registers 11-3
  - executable files
    - viewing 2-31
  - executing an image
    - armsd 4-8
    - from ADW 3-22
    - from APM 2-18
  - Execution profile 8-19
  - Execution regions 14-3
  - Execution Window 3-12
  - exiting
    - ARM Debugger for Windows 3-27
    - ARM Project Manager 2-19
    - armsd 4-9
  - Expression Window 3-17
- ## F
- files
    - adding to a project 2-15
    - assigned to multiple partitions 2-29
    - building 2-30
  - finding errors 2-17
  - FIQ
    - exception 11-2
  - handlers 11-6
  - flash downloading
    - ADW 3-45
    - Angel 6-6, 6-24
  - floating-point 15-2
    - ARM software library 15-3
    - binary data operations 15-11
    - calling from assembler 15-4
    - comparisons 15-10
    - controlling exceptions 15-15
    - data transfers 15-8
    - EmbeddedICE 7-39
    - hardware and software 15-2
    - instruction set 15-8
    - multiple data transfers 15-9
    - register transfers 15-9
    - unary data operations 15-12
  - floating-point calls
    - assembler 15-4
  - force building a project 2-5, 2-22
  - formats
    - display 3-36
  - frame pointers
    - Thumb 12-10
  - Function Names Window 3-17
  - functions
    - ARM and Thumb copies 12-15
    - stepping through 3-23
    - stepping through (armsd) 4-9
  - fusion stack 6-23
- ## G
- global variables
    - examining 3-26
  - Globals Window 3-18
- ## H
- halfwords
    - reducing code size 8-16
  - handling SWIs
    - in Thumb state 11-33

- Hello World example
  - APM 2-13
  - command-line development 4-2
- help
  - armsd 4-7
- high-level symbols 3-7

## I

- images
  - building 2-16
  - debugging 2-19
  - displaying information about 3-34
  - executing 2-18, 3-22
  - executing (armsd) 4-8
  - larger than expected 13-27
  - loading 3-22
  - reducing size 8-16
  - reloading 3-27
  - stepping through 3-23
  - variables (armsd) 4-10
- immediate evaluation of variables 3-35
- indirection
  - adw 3-37
- initialization
  - no code present 13-27
  - on RESET 13-3
- input patterns 2-41
- installing
  - exception handlers 11-8
- instruction sets
  - ARM core 9-3
  - Thumb 9-4
- instruction tracing 8-25
- integer
  - display 3-37
- integer-like structures 10-12
- inter-link-unit 10-4
- interrupt handlers 11-19
- interworking
  - ARM and Thumb 12-10
  - assembler 12-18
- IRQ
  - exception 11-2

- handlers 11-6

## J

- JTAG
  - and Angel 6-28
  - protocol conversion unit 7-3
  - signal connection 7-20
- jump table
  - example 9-23
- jumps
  - and code speed 8-18

## L

- Late Debugger Startup 6-14
- leaf functions
  - Thumb 12-10
- libraries
  - embedded C 13-16
- lines of code
  - stepping to 3-23
- link and compile
  - separating 4-4
- linking
  - libraries (armlink) 4-4
- little-endian
  - Angel 6-26
- load agent 6-21
- load regions 14-3
- loading
  - constants 9-10
  - images 3-22
- local variables
  - examining 3-26
- Locals Window 3-18
- Low Level Symbols Window 3-19
- low-level symbols 3-7
  - breakpoints 3-40

## M

- Macrocell
  - accessing 7-28
- map files 8-8
  - armsd.map 8-8
  - Dhrystone example 8-12
  - format 8-9
- memory
  - copying from disk 3-42
  - displayed 3-37
  - displaying as disassembly 3-37
  - examining 3-27
  - saving to disk 3-42
  - simulating in map file 8-8
- Memory Window 3-19
- menus
  - ADW 3-12
- models
  - ARMulator 5-2
  - dummy MMU 5-11
  - profiler 5-12
  - tracer 5-13
  - watchpoints 5-15
  - Windows Hourglass 5-15
- modifying
  - debugger variables (armsd) 4-10
  - registers 3-26
  - variables 3-26
- moving projects
  - force build 2-22
- multiplication
  - returning a 64-bit result 10-14

## N

- new features 1-5
- non integer-like structures 10-13

## O

- object files
  - viewing 2-31

- output patterns 2-41
- Overlay Format 14-2
- OVERLAY option 14-15
- overlays 14-15
  - assigning AREAs 14-17
  - check\_for\_invalidated\_returns 14-25
  - clash detection 14-29
  - description for the linker 14-17
  - dynamic 14-16
  - Image\$\$load\_seg 14-21
  - intercepting returns to overwritten segments 14-23
  - load\_seg\_and\_go 14-20
  - load\_seg\_and\_ret 14-20, 14-25
  - load\_segment 14-20, 14-22
  - manager 14-19, 14-29
  - OVERLAY option 14-15
  - PCIT structure 14-20
  - relocation directives 14-16
  - segment clash 14-29
  - static 14-15
  - using scatter loading 14-28

## P

- partitions 2-6
  - creating 2-41
  - files assigned to 2-29
- passing structures 10-11
- paths
  - editing for tools 2-37
- PC sampling 8-20
- PCIT section 14-20
- performance
  - improving 8-14
  - measuring 8-6
- PID board
  - Angel 6-23
- pipeline refill 9-16
- prefetch abort
  - exception 11-2
  - exception handler 11-27
  - handler 11-27
  - returning from 11-6

- processors
  - clock speeds 8-11
  - responding to exceptions 11-5
- profiler 5-12
- profiling 8-19
  - ARM Debugger for Windows 3-8
  - data
    - collecting 8-20
    - collecting in ADW 3-44
    - creating report 8-21
  - instruction tracing
    - ARMulator 8-25
  - restrictions 3-8
  - sorts example 8-23
- project hierarchy 2-5
- project output 2-5
  - changing name 2-24
  - using 2-18
- project templates 2-6, 2-33
  - creating 2-39
  - creating a project from 2-34
  - editing 2-38
  - editing details 2-40
  - modifying 2-34
  - supplied with APM 2-34
- project view 2-9
- Project Window 2-9
- projects 2-3
  - adding files 2-15
  - adding variants 2-22
  - building 2-16
  - changing name 2-7, 2-24
  - converting 2-25
  - correcting problems 2-17
  - creating 2-13, 2-34
  - viewing 2-15
- properties
  - variable 3-37

## R

- RAM
  - measuring requirements 8-4
- RDI Log Window 3-19

- RDI Protocol Log
  - displaying 3-38
- real-time simulation 8-8
- registers
  - containing addresses 9-19
  - examining 3-26
  - exceptions 11-3
  - modifying 3-26
  - usage 10-8
- Registers Window 3-19
- regular expressions 3-8
- reloading images 3-27
- relocation directives
  - dynamic overlays 14-16
- Remote Debug Information
  - displaying 3-38
- Remote Debugging Interface 3-9
- Remote\_A 3-4
  - channel viewers in ADW 3-31
  - configuring in ADW 3-31
- Remote\_D 3-5
  - configuring in ADW 3-32
- removing
  - breakpoints 3-25
  - breakpoints (armsd) 4-7
  - watchpoints 3-25, 4-8
- reset 7-20
  - applications entered via 13-3
  - EmbeddedICE 7-25
  - exception 11-2
  - faking 7-24
  - handler 11-25
- reset exception handler 11-25
- return address 11-6
- return instruction 11-6
- returning structures 10-11
- ROM
  - at 0 systems
    - debugging with EmbeddedICE 7-27
  - building with scatter loading 13-15
  - C libraries 13-16
  - code entered at base address 13-5
  - code loaded at address 0 13-14
  - debugging systems with EmbeddedICE 7-24

- measuring requirements 8-4
- stack checking code 13-24
- start-up for applications 13-3
- troubleshooting code 13-24

## S

- S option
  - armcc 4-5
- scatter loading 14-3
  - description format 14-5
  - format 14-2
  - image formats 14-4
  - overlays 14-28
- search paths 3-9
  - adding 3-34
- Search Paths Window 3-19
- segment clash
  - overlays 14-29
- semihosting 7-17
- setjmp()
  - and code speed 8-18
- setting
  - breakpoints (armsd) 4-7
  - simple breakpoint 3-24
  - simple watchpoints 3-25
  - watchpoints (armsd) 4-8
- shorts
  - reducing code size 8-16
- simulated time
  - reducing 8-13
- simulation
  - real-time 8-8
- single stepping
  - through an image (ADW) 3-23
  - through source (armsd) 4-9
- software interrupt
  - exception 11-2
- sorts
  - profiling example 8-23
- source code
  - displaying (armsd) 4-10
- Source File Window 3-20
- source files

- assigning to a partition 2-41
- building 2-30
- creating 2-14, 2-28
- displaying 3-34
- listing 3-35
- search paths 3-9
- Source Files List Window 3-20
- specifying a disassembly mode 3-38
- stack checking
  - ARM and Thumb 12-10
  - ROM code 13-24
- static overlays 14-15
- status area 2-10
- stepping
  - through an image 3-23
  - through source (armsd) 4-9
- stopping a build 2-25
- strings
  - copying using assembler 9-25
- structure passing and returning 10-11
- sub-projects 2-3
- SWI exception handler 11-12
- SWIs
  - and undefined instruction handlers 11-6
  - handlers 11-12
  - handling in Thumb state 11-33
  - handling with EmbeddedICE 7-18
- symbols 3-7
  - breakpoints 3-40

## T

- target environments
  - configuring 3-28
  - configuring ARMulator 3-30
  - configuring EmbeddedICE 3-33
  - configuring Remote\_A 3-31
  - configuring Remote\_D 3-32
  - debugging 3-3
- tasm 1-3
- tcc 1-3
- templates 2-6, 2-33
  - creating 2-39
  - editing 2-38

- modifying 2-34
- Thumb 12-18
  - assembler module 9-8
  - C libraries 12-15
  - code 3-6
  - frame pointers 12-10
  - instruction set 9-4
  - veneers 12-10
- Thumb state
  - and ARM state 12-2
  - exception handler 11-31
- Thumb-aware exception handler 11-31
- timer accuracy
  - and EmbeddedICE 7-32
- toggle interleaving
  - disassembled code 3-37
- tools
  - assigning to a build step 2-42
  - configuring 2-20
  - editing paths 2-37
  - resetting configuration 2-21
- TPCS 10-9
  - register names and usage 10-9
- tracer 5-13
- tracing 8-25

## U

- undefined instruction
  - exception 11-2
  - handler 11-26
- user mode 11-3

## V

- variables
  - ADW
    - changing display formats 3-36
    - displaying 3-15, 3-35
    - examining 3-26
    - immediate evaluation 3-35
    - modifying 3-26
    - properties 3-37

- window 3-18
- APM 2-7
  - build step patterns 2-41
  - editing 2-36
- armsd
  - displaying 4-10
  - image 4-10
  - modifying 4-10
- variants 2-6
  - adding to a project 2-22
  - building 2-23
- vector table 11-3
- veneers
  - ARM and Thumb code 12-10
  - assembler and Thumb 12-18
- View Window 2-12
- viewing
  - binaries 2-12, 2-31
  - build logs 2-10
  - build messages 2-10
  - debugger variables (armsd) 4-10
  - dependencies 2-10
  - disassembled code 3-37
  - object and executable files 2-31
  - projects 2-15
  - Remote Debug Information 3-38
  - source code (armsd) 4-10
  - source files in ADW 3-34
  - variable properties 3-37
  - variables 3-35
  - variants 2-10

## W

- watchpoints 3-9, 5-15
  - and EmbeddedICE 7-13
  - armsd 4-8
  - complex 3-40
  - editing 3-41
  - removing 3-25
  - simple 3-25
- Watchpoints Window 3-20
- windows
  - ARM Debugger for Windows 3-11

# Index

---

- ARM Project Manager 2-9
- Windows Hourglass 5-15
- window-specific menus 3-12







<http://www.arm.com>

---

#### **ENGLAND**

Advanced RISC Machines Limited  
Fulbourn Road, Cherry Hinton  
Cambridge CB1 4JN  
UK  
Telephone: +44 1223 400400  
Facsimile: +44 1223 400410  
Email: [info@armltd.co.uk](mailto:info@armltd.co.uk)

#### **JAPAN**

Advanced RISC Machines K.K.  
KSP West Bldg, 3F 300D, 3-2-1 Sakado  
Takatsu-ku, Kawasaki-shi  
Kanagawa, 213 Japan  
Telephone: +81 44 850 1301  
Facsimile: +81 44 850 1308  
Email: [info@armltd.co.uk](mailto:info@armltd.co.uk)

#### **GERMANY**

Advanced RISC Machines Limited  
Otto-Hahn Str. 13b  
85521 Ottobrunn-Riemerling  
Munich, Germany  
Telephone: +49 89 608 75545  
Facsimile: +49 89 608 75599  
Email: [info@armltd.co.uk](mailto:info@armltd.co.uk)

#### **USA**

ARM USA Incorporated  
Suite 5  
985 University Avenue  
Los Gatos, CA 95030 USA  
Telephone: +1 408 399 5199  
Facsimile: +1 408 399 8854  
Email: [info@arm.com](mailto:info@arm.com)